

Join the discussion! p2p.wrox.com



Wrox Programmer to Programmer™

Shell Scripting: Expert Recipes for Linux,
Bash, and More

Shell脚本编程诀窍 ——适用于Linux、Bash等



[英] Steve Parker 著
万 千 译

清华大学出版社

这些实用脚本是经过实践检验的，可以快速应用或很容易经过调整满足自己的需求

shell是与Unix和Linux系统通信的主要方法，并通过使简单到中间任务的自动化，提供了一种直接的编程手段。在《Shell脚本编程诀窍——适用于Linux、Bash等》中，精通Unix、Linux与shell脚本编程的专家Steve Parker分享了一组shell实用脚本编程诀窍。这些实用脚本编程诀窍可以拿来直接使用，或者很容易对其修改以适应各种环境与条件。本书一开始介绍了一些理论与原理，并且每个讨论的话题都包含了具有深刻意义的示例。介绍完理论之后开始深入讨论shell编程，内容涵盖所有的Unix种类，但主要集中于Linux与Bash shell。至始至终，本书都在介绍一些可信的、用于实际用途的实用脚本编程诀窍，以及用来快速上手的工具。

主要内容

- ◆ 本书汇总了许多很有用的shell实用脚本编程诀窍，能用来处理现实中的各种问题
- ◆ 本书包含的实用脚本编程诀窍使用了文件与文本控制，以及通用的系统管理员任务
- ◆ 本书提供的实用脚本编程诀窍随时可使用或修改
- ◆ 本书讨论了变量、if/then条件、循环、函数、管道与重定向等

作者简介

Steve Parker是专门从事Solaris与GNU/Linux方面的IT咨询师。他已经提供了超过10年的咨询服务。他创办著名的*Bourne Shell Programming/Scripting Tutorial*(<http://steve-parker.org/sh/sh.shtml>)。该网站每年有超过一百万的访问量。



wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

清华大学出版社数字出版网站

WQBook 书文泉
www.wqbook.com

Wrox
An Imprint of
WILEY

ISBN 978-7-302-29781-9



9 787302 297819 >

定价：68.00元

Shell 脚本编程诀窍

——适用于 Linux、Bash 等

[英] Steve Parker 著
万 千 译

清华大学出版社

北 京

Steve Parker

Shell Scripting: Expert Recipes for Linux, Bash, and More

EISBN: 978-1-118-02448-5

Copyright © 2011 by Steve Parker, Manchester, England

All Rights Reserved. This translation published under license.

本书中文简体字版由 John Wiley & Sons, Inc. 授权清华大学出版社出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2011-6438

本书封面贴有 Wiley 公司防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

Shell 脚本编程诀窍——适用于 Linux、Bash 等/(英)帕克(Parker, S.) 著; 万千 译. —北京: 清华大学出版社, 2012.9

书名原文: Shell Scripting: Expert Recipes for Linux, Bash, and More

ISBN 978-7-302-29781-9

I. ①S… II. ①帕… ②万… III. ①UNIX 操作系统—程序设计 IV. ①TP316.81

中国版本图书馆 CIP 数据核字(2012)第 189681 号

责任编辑: 王 军 于 平

装帧设计: 牛艳敏

责任校对: 蔡 娟

责任印制: 何 莘

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座 邮 编: 100084

社 总 机: 010-62770175 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者: 清华大学印刷厂

装 订 者: 三河市新茂装订有限公司

经 销: 全国新华书店

开 本: 185mm×260mm

印 张: 34.5

字 数: 840 千字

版 次: 2012 年 9 月第 1 版

印 次: 2012 年 9 月第 1 次印刷

印 数: 1~3000

定 价: 68.00 元

产品编号: 043826-01

作者简介

Steve Parker 是具有 20 年 Unix 经验与 15 年 GNU/Linux 经验的 Unix 与 Linux 顾问。他编写了在线 shell 脚本编程教程并对其进行维护(<http://steve-parker.org/sh/sh.shtml>)。

Steve 提供 IT 咨询服务，还提供 shell 脚本编程与 Unix、Linux 技术的培训课程。可以通过 <http://sgpit.com/> 与他联系。

技术编辑简介

John Kennedy 自 1997 年开始就一直作为系统管理员来使用 Linux(与 Unix)。他使用过 Red Hat、SUSE、Debian、Ubuntu、Solaris 与 HP-UX。他在 2000 年开始使用 bash 编程，因为他感到工作繁重，需要一些工具来为他完成这些琐碎的任务。

在学习 Linux 与 Unix 之前，John 在美国空军当了 9 年的通信系统操作人员，并辗转于德国、德克萨斯州与英格兰之间。离开军队后，他曾在内布拉斯加州与宾夕法尼亚州生活过，现在回到了英格兰。

John 现在的工作是为伦敦的一家媒体公司当架构工程师。他与妻子 Michele 和儿子 Kieran 居住在牛津附近，他的女儿 Denise 刚刚在美国完成了大学学业。

当 John 不面对电脑时，他喜欢陪儿子看(英式)足球以及享受幸福的家庭生活。

致 谢

如果没有 Wiley 员工给予的帮助就不会有本书的完成。出版过程的每一步都是我从未经历过的，Christina Haviland 在每一步都进行了很好的指导。John Kennedy 在整个过程中提供反馈并予以鼓励。Nancy Rapoport 对本书的细节进行了仔细的审查。

从个人的角度出发，我要感谢那些来自 Acorn、Sinclair 以及其他一些 20 世纪 80 年代早期的公司的人们，因为是他们生产了廉价的、能让孩子们真正用来学习编程的计算机。还要感谢 BBC 在其 BBC Micro 项目中的前瞻性，还有该项目中的 TV 计划，以及他们从事的开发工作。下一代需要像 BBC Micro 这样的项目；不需要使用奇怪的 IDE 为手机编写应用程序，而是直接对真正的系统进行操作。值得称颂的是 Arduino 项目从硬件级别来开始这项工作。这是个不错的项目，它使得对硬件的研究变得简单，而不需要有个知识渊博的叔叔在一旁翻译寄存器的值。自由软件，尤其是最近出现的谷歌代码之夏，这都是新一代全新开发的理想起点。有个比较令人不安的观点是，新一代的人只知道使用设备，却不知道如何进行开发。上面提到的这些项目为将来带来了希望。

我还要感谢 ICL。在那里我认识了 Douglas、Capitan、Jit 与 Ketan。我们对 DRS/NX 进行了测试，并可以直接跟用户空间内核开发人员进行交流。这是非常罕见的待遇，并且在那里我喜欢上了 Unix。还要感谢的是过去当 Usenet 还可读时那些经常出现在 comp.unix.shell 上的朋友们；是你们教会了我这么多，而以前的我肯定显得很无知。

ICL 雇我做内核与用户控件开发人员，而每个 GNU/Linux 操作系统用户都可以获取到我在那里学到的东西。在撰写本书的过程中，我得以引用一些我未曾谋面(而且可能永远不会遇见)的人的电子邮件内容，其中是关于 Unix、Linux 与 shell 功能的讨论。类似地，在专业的环境下，我有幸与特定 Linux 内核功能的开发人员进行在线交谈，讨论他们在不同版本 Linux 内核中的实现方式。如果采用不同的开发模式，则对一个版本而言都是不可能的。同样地，bash shell 的维护者 Chet Ramey 对 bash 的实现细节进行了一些邮件回复。

从专业与 IT 社区的角度出发，我要感谢 Ken Thompson、Dennis Ritchie、Brian Kernighan、Doug McIlroy、David Korn 与 Steve Bourne(仅列出这几个人名)对 C 语言、Unix 以及公认的开发环境的贡献。这些有远见的人提出的概念持续了 40 多年。

我还要感谢 Richard M. Stallman 博士为世界带来的 GNU 项目、GPL 以及自由软件基金会，还有他为倡导软件自由而奉献了自己的一生。世界需要理想主义者，Stallman 就是其中之一。我相信历史将证明他的理念是正确的，即软件应当共享而不是相互隐藏。这就是科学的传统，而且如果把计算机科学作为严肃的科学来看待的话，这一传统也将适用。

感谢所有 GNU 程序员与 Linux 内核开发人员，因为他们将所有这些自由软件组合成各种各样不同、可用且自由的操作系统。还要感谢 Unix 开发人员，因为他们为不同的雇主按照各种许可证编写代码。

最后，我要感谢 Bill Joy、Vinod Khosla、Andy Bechtolsheim 与 Scott McNealy 对于 Sun Microsystems 与 Solaris 操作环境所做的贡献。另外要感谢 Jonathan Schwartz 将公司的大多数软件开源化(为了将 OpenOffice.org 开源甚至将 StarDivision 买下来)，感谢 JDS 在业界一度不理解 GNOME 项目的模型时对 GNOME 项目的贡献。

前言

shell 是每个 Unix 与 Linux 系统的标准接口。用户与管理员都有使用 shell 的经验，而且将命令组合到 shell 脚本中是很自然的做法。然而，这只是冰山一角。

shell 实际上是一个完整的编程语言。它具有变量、函数以及如数组(包括关联数组)这样更高级的数据结构。因为直接连接到内核，所以 shell 具有内嵌到语法中的原生文件 I/O 以及进程与作业控制。Unix 比较知名的主要特征都可以在 shell 中找到，而且可以在 shell 脚本中使用。

编写本书是为了对 shell 进行较为全面的介绍，并且无论用户具有何种背景与经验都能从本书中得到一些收获。本书主要面向中级与高级 Unix 与 Linux 管理员，以及可能感兴趣的其他高级用户。本书假设读者至少会用一种 Unix 系统，并且可能已经编写了一些 shell 脚本，但希望提高自身脚本编写的水平。

有经验的读者可以掠过前两章。更高级的用户可以掠过前 4 章，尽管其中包含了一些值得回顾的细节。

本书内容

本书介绍 shell 脚本编程，主要针对 Bourne shell 与 POSIX 兼容的 shell，但也广泛涵盖了新近的一些发展情况，尤其是 bash shell。bash shell 几乎总是会包含在 GNU/Linux 操作系统中，也包含在了大多数商业 Unix 中。另外，KornShell 也被广泛用于大部分这样的闭源或开源操作系统中。

本书结构

本书分为 4 个部分。第 I 部分介绍 shell 的基本功能和语法；第 II 部分介绍 shell 脚本可以使用的工具；第 III 部分给出了一些涵盖更广泛话题的实用脚本；第 IV 部分是参考信息。

第 I 部分在 4 个部分中是最长的；它介绍变量、通配符、条件执行、循环、函数、数组与进程。理论的介绍是通过很多示例对所讲内容的演示来完成的。这些脚本中很多都相当简单，因为它们只集中于 shell 的某一方面。

第 II 部分介绍 shell 外部的一些工具。这些工具使 shell 脚本的功能更加强大。这一部分分为 3 章，分别介绍文本、文件与通用的系统管理。第 II 部分中的例子本身要更加接近现实情况一些，并且比第 I 部分的脚本更长、更复杂。

第 III 部分是一个 shell 编程实用脚本的集合。我们希望读者能自己运用这些脚本，但选

择这些脚本的原因也是出于对前两部分介绍的内容的演示。这些脚本还包含了能用在现实 shell 脚本中的各种不同的方法与技术。本书的这一部分是对真实的现实问题进行处理，并且没有对特定问题进行解释。通过执行必要的操作，这些脚本实现的任务有安装初始化脚本、编写彩色的实时交互式游戏、分析 HTML、控制进程、将脚本翻译成多种语言、编写 CGI 脚本、创建图形化报告等。

第 IV 部分列出了一些补充材料的链接以及术语表。

使用前的准备

第 2 章介绍了一些可选的方法。利用这些方法可以获取到自己的 shell 环境，并为用户对其进行量身配置。在活动系统上进行实验是一个选择，但不是较好的选择。更好的方法是建立一个测试账户。在专门的测试机器或虚拟机上运行则再好不过。像 VirtualBox 或 VMWare Player 这样的虚拟化软件可以免费获取。利用它们可以对最有风险的 root 用户脚本进行无风险测试。

源代码

在练习书中的示例时，可以选择手动输入代码或者使用本书附带的源代码文件。书中用到的所有源代码都可以从 www.wrox.com 下载。进入站点 <http://www.wrox.com> 后，只需要找到本书的书名(使用 Search 搜索框或书名列表)，单击本书详细信息页面上的 Download Code 链接，就可以得到本书所有的源代码。



因为很多书的书名都相似，所以用 ISBN 搜索更为容易。本书英文版的 ISBN 是 978-1-118-02448-5。

下载完代码后，用您喜欢的压缩工具把它解压缩。此外，也可以去 Wrox 的主下载页面 www.wrox.com/dynamic/books/download.aspx 找到本书或 Wrox 出版的其他书籍的代码。

勘误表

尽管我们竭尽所能来确保在正文和代码中没有错误，但人无完人，错误难免会发生。如果您在 Wrox 出版的书中发现了错误(例如拼写错误或代码错误)，我们将非常感谢您的反馈。发送勘误表将节省其他读者的时间，同时也会帮助我们提供更高质量的信息。

要找到本书的勘误表页面，可以进入 www.wrox.com，使用 Search 搜索框或书名列表定位本书，然后在本书的详细信息页面上单击 Book Errata 链接。在这个页面上可以查看为本书提交的、Wrox 编辑粘贴上去的所有错误。完整的书名列表(包括每本书的勘误表)

也可以从 www.wrox.com/misc-pages/booklist.shtml 上获得。

如果您在本书的勘误页面上没有看到您发现的错误，可以到 www.wrox.com/contact/techsupport.shtml 上填写表单，把您发现的错误发给我们。我们会检查这些信息，如果属实，就把它添加到本书的勘误页面上，并在本书随后的版本中更正错误。

p2p.wrox.com

如果想和作者或同行进行讨论，请加入 <http://p2p.wrox.com> 上的 P2P 论坛。该论坛是一个基于 Web 的系统，您可以发布有关 Wrox 图书及相关技术的消息，与其他读者或技术人员交流。该论坛提供了订阅功能，当您感兴趣的主题有新帖子发布时，系统会邮件通知。Wrox 的作者、编辑、其他业界专家和像您一样的读者都会出现在这些论坛中。

在 <http://p2p.wrox.com> 网站上，您会找到很多不同的论坛，它们不但有助于您阅读本书，还有助于您开发自己的应用程序。加入论坛的步骤如下：

- (1) 进入 <http://p2p.wrox.com>，单击 Register 链接。
- (2) 阅读使用条款，然后单击 Agree 按钮。
- (3) 填写加入该论坛必需的信息和其他您愿意提供的信息，单击 Submit 按钮。
- (4) 您将收到一封电子邮件，描述如何验证您的账户和完成加入过程。



不加入 P2P 也可以阅读论坛里的消息。但是如果要发布自己的消息，就必须加入。

加入之后，就可以发布新的消息和回复其他用户发布的消息。可以随时在 Web 上阅读论坛里的消息。如果想让某个论坛的新消息以电子邮件的方式发给您，可以单击论坛列表中论坛名称旁边的 **Subscribe to this Forum** 图标。

要了解如何使用 Wrox P2P 的更多信息，请阅读 P2P FAQ，其中回答了论坛软件如何使用的问题，以及许多与 P2P 和 Wrox 图书相关的问题。要阅读 FAQ，单击任何 P2P 页面上的 FAQ 链接即可。

目 录

第 I 部分 基本概念

第 1 章	Unix、GNU 和 Linux 的历史	3
1.1	Unix	3
1.1.1	“一切皆文件”与管道	5
1.1.2	BSD	6
1.2	GNU	7
1.3	Linux	10
1.4	本章小结	12
第 2 章	环境的搭建	13
2.1	操作系统	13
2.1.1	GNU/Linux	13
2.1.2	BSD	15
2.1.3	商业 Unix	15
2.1.4	Microsoft Windows	15
2.2	编辑器	16
2.2.1	图形化文本编辑器	16
2.2.2	终端模拟器	19
2.2.3	非图形化文本编辑器	19
2.3	系统环境的搭建	21
2.3.1	shell 配置文件	21
2.3.2	别名	23
2.3.3	vim 设置	27
2.4	本章小结	28
第 3 章	变量	29
3.1	使用变量	29
3.1.1	类型	30
3.1.2	变量的赋值	30
3.1.3	位置参数	34
3.1.4	返回码	38
3.1.5	删除变量	41

3.2	预定义变量和标准变量	42
3.2.1	BASH_ENV	43
3.2.2	BASHOPTS	43
3.2.3	SHELLOPTS	44
3.2.4	BASH_COMMAND	46
3.2.5	BASH_SOURCE、 FUNCNAME、LINENO 和 BASH_LINENO	47
3.2.6	SHELL	51
3.2.7	HOSTNAME 和 HOSTTYPE	51
3.2.8	工作目录	51
3.2.9	PIPESTATUS	51
3.2.10	TIMEFORMAT	52
3.2.11	PPID	53
3.2.12	RANDOM	54
3.2.13	REPLAY	54
3.2.14	SECONDS	55
3.2.15	BASH_XTRACEFD	55
3.2.16	GLOBIGNORE	57
3.2.17	HOME	58
3.2.18	IFS	58
3.2.19	PATH	59
3.2.20	TMOUT	60
3.2.21	TMPDIR	61
3.2.22	用户标识变量	61
3.3	本章小结	62
第 4 章	通配符扩展	63
4.1	文件名扩展(globbing)	63
4.1.1	bash 的文件名扩展特性	66
4.1.2	shell 选项	67
4.2	正则表达式和引用	71

4.2.1 正则表达式概述	72	7.1.6 使用模式剪裁字符串	143
4.2.2 引用	73	7.2 字符串查找	147
4.3 本章小结	77	7.2.1 查找与替换	147
第 5 章 条件执行	79	7.2.2 模式替换	149
5.1 if/then	79	7.2.3 模式删除	149
5.2 else	80	7.2.4 大小写转换	149
5.3 elif	81	7.3 提供默认值	150
5.4 test(I)	83	7.4 间接操作	153
5.4.1 测试标志	84	7.5 使用 source 命令加载变量	155
5.4.2 文件比较测试	91	7.6 本章小结	156
5.4.3 字符串比较测试	92	第 8 章 函数和库	157
5.4.4 正则表达式测试	94	8.1 函数	157
5.4.5 数值测试	97	8.1.1 函数定义	157
5.4.6 组合测试	98	8.1.2 函数输出	158
5.5 case	101	8.1.3 写入文件	160
5.6 本章小结	105	8.1.4 整个函数的输出重定向	163
第 6 章 使用循环进行流控制	107	8.1.5 函数陷阱	167
6.1 for 循环	107	8.1.6 递归函数	168
6.1.1 for 循环的使用时机	108	8.2 变量的作用域	173
6.1.2 向 for 提供数据	108	8.3 库	177
6.1.3 C 风格的 for 循环	114	8.3.1 库的创建与访问	179
6.2 while 循环	115	8.3.2 库的结构	179
6.2.1 while 循环的使用时机	116	8.3.3 网络配置库	183
6.2.2 while 循环的用法	116	8.3.4 库的使用	187
6.3 嵌套循环	122	8.4 getopt	187
6.4 循环的退出与继续	122	8.4.1 错误处理	190
6.5 带 case 的 while 循环	126	8.4.2 函数中的 getopt	191
6.6 until 循环	127	8.5 本章小结	194
6.7 select 循环	129	第 9 章 数组	195
6.8 本章小结	134	9.1 数组的赋值	195
第 7 章 变量(续)	135	9.1.1 一次一个	196
7.1 变量的用法	135	9.1.2 一次全部	196
7.1.1 变量的类型	137	9.1.3 按索引	197
7.1.2 变量的长度	138	9.1.4 从源中一次全部读取	197
7.1.3 特殊字符串操作符	140	9.1.5 从输入读取	199
7.1.4 按照长度剪裁变量字符串	141	9.2 数组的访问	201
7.1.5 从字符串末尾剪裁	143	9.2.1 用索引访问	201

9.2.2 数组的长度	202	10.9.7 sysctl	250
9.2.3 用变量索引访问	203	10.10 本章小结	250
9.2.4 从数组中选择元素	205	第 11 章 shell 的选择与使用	251
9.2.5 显示整个数组	206	11.1 Bourne shell	251
9.3 关联数组	206	11.2 Kornshell	252
9.4 数组操作	207	11.3 C shell	252
9.4.1 数组的复制	207	11.4 Tenex C shell	252
9.4.2 向数组追加元素	209	11.5 Z shell	253
9.4.3 从数组中删除元素	211	11.6 Bourne Again Shell	253
9.5 高级技术	212	11.7 Debian Almquist Shell	253
9.6 本章小结	213	11.8 点文件	254
第 10 章 进程	215	11.8.1 交互式登录 shell	255
10.1 ps 命令	215	11.8.2 交互式非登录 shell	256
10.1.1 ps 显示的行宽	216	11.8.3 非交互式 shell	257
10.1.2 精确分析进程表	217	11.8.4 登出脚本	257
10.2 killall	219	11.9 命令提示符	257
10.3 /proc 虚拟文件系统	220	11.9.1 PS1 提示符	257
10.4 prtstat	221	11.9.2 PS2、PS3 和 PS4 提示符	259
10.5 I/O 重定向	222	11.10 别名	260
10.5.1 向已有文件追加输出	224	11.10.1 节省时间	260
10.5.2 重定向的权限	225	11.10.2 修改行为	261
10.6 exec	225	11.11 history 命令	262
10.6.1 使用 exec 替换已有 程序	225	11.11.1 回调命令	262
10.6.2 使用 exec 修改重定向	226	11.11.2 搜索历史	263
10.7 管道	233	11.11.3 时间戳	263
10.8 后台处理	233	11.12 Tab 补全	265
10.8.1 wait 命令	234	11.12.1 ksh	265
10.8.2 使用 nohup 防止进程 挂起	235	11.12.2 tcsh	266
10.9 /proc 和/sys 的其他特性	238	11.12.3 zsh	266
10.9.1 /proc/version	238	11.12.4 bash	267
10.9.2 SysRq	238	11.13 后台、前台与作业控制	268
10.9.3 /proc/meminfo	240	11.13.1 后台进程	268
10.9.4 /proc/cpuinfo	241	11.13.2 作业控制	268
10.9.5 /sys	241	11.13.3 nohup 和 disown	271
10.9.6 /sys/devices/system/node	248	11.14 本章小结	272

第 II 部分 系统工具使用与扩展诀窍

第 12 章 文件操作	275
12.1 stat.....	275
12.2 cat.....	277
12.2.1 行号标记.....	277
12.2.2 处理空白行.....	278
12.2.3 非打印字符.....	279
12.3 cat 的反转词 tac.....	280
12.4 重定向.....	281
12.4.1 重定向输出: 单个 大于符号(>).....	281
12.4.2 追加: 双大于符号(>>).....	282
12.4.3 输入重定向: 单个 小于符号(<).....	284
12.4.4 here 文档: 双小于 符号(<< EOF).....	286
12.5 dd.....	289
12.6 df.....	291
12.7 mktemp.....	292
12.8 join.....	293
12.9 install.....	294
12.10 grep.....	296
12.10.1 grep 标志.....	297
12.10.2 grep 正则表达式.....	298
12.11 split.....	299
12.12 tee.....	301
12.13 touch.....	302
12.14 find.....	303
12.15 find -exec.....	306
12.16 本章小结.....	310
第 13 章 文本操作	311
13.1 cut.....	311
13.2 echo.....	312
13.2.1 dial1 脚本.....	312
13.2.2 dial2 脚本.....	315
13.3 fmt.....	316
13.4 head 和 tail.....	319

13.4.1 奖牌脚本.....	319
13.4.2 世界杯脚本.....	320
13.5 od.....	324
13.6 paste.....	328
13.7 pr.....	331
13.8 printf.....	332
13.9 shuf.....	334
13.9.1 掷骰子.....	334
13.9.2 发牌.....	335
13.9.3 旅行线路.....	336
13.10 sort.....	338
13.10.1 按照键进行排序.....	338
13.10.2 按照日期与时间对 日志文件排序.....	340
13.10.3 对人类可读的数值 进行排序.....	342
13.11 tr.....	343
13.12 uniq.....	346
13.13 wc.....	348
13.14 本章小结.....	349
第 14 章 系统管理工具	351
14.1 basename.....	351
14.2 date.....	353
14.2.1 date 的典型用法.....	353
14.2.2 date 的一些更有趣的 用法.....	357
14.3 dirname.....	358
14.4 factor.....	360
14.5 id、groups 与 getent.....	362
14.6 logger.....	365
14.7 md5sum.....	366
14.8 mkfifo.....	368
14.8.1 主与从.....	369
14.8.2 颠倒顺序.....	371
14.9 联网.....	373
14.9.1 telnet.....	373
14.9.2 netcat.....	374
14.9.3 ping.....	376

14.9.4	编写 ssh 与 scp 脚本	378
14.9.5	OpenSSL	381
14.10	nohup	387
14.11	seq	388
14.11.1	整数序列	389
14.11.2	浮点数序列	391
14.12	sleep	391
14.13	timeout	392
14.13.1	关闭脚本	394
14.13.2	网络超时	396
14.14	uname	398
14.15	uuencode	399
14.16	xargs	400
14.17	yes	403
14.18	本章小结	404

第III部分 系统管理的实用脚本

第 15 章 shell 特性.....407

15.1	实用脚本 15-1: 安装 初始化脚本	407
15.1.1	用到的技术	407
15.1.2	概念	408
15.1.3	潜在的陷阱	408
15.1.4	脚本结构	408
15.1.5	脚本代码	410
15.1.6	调用结果	411
15.1.7	小结	412
15.2	实用脚本 15-2: RPM 报告	412
15.2.1	用到的技术	412
15.2.2	概念	412
15.2.3	潜在的陷阱	413
15.2.4	脚本结构	413
15.2.5	脚本代码	414
15.2.6	调用结果	417
15.2.7	小结	418
15.3	实用脚本 15-3: postinstall 脚本	418
15.3.1	用到的技术	418

15.3.2	概念	419
15.3.3	潜在的陷阱	419
15.3.4	脚本结构	420
15.3.5	脚本代码	421
15.3.6	调用结果	423
15.3.7	小结	423

第 16 章 系统管理

16.1	实用脚本 16-1: 初始化脚本	425
16.1.1	用到的技术	426
16.1.2	概念	426
16.1.3	潜在的陷阱	427
16.1.4	脚本结构	428
16.1.5	脚本代码	429
16.1.6	调用结果	430
16.1.7	小结	431
16.2	实用脚本 16-2: CGI 脚本	431
16.2.1	用到的技术	431
16.2.2	概念	431
16.2.3	潜在的陷阱	432
16.2.4	脚本结构	433
16.2.5	脚本代码	436
16.2.6	调用结果	439
16.2.7	小结	442
16.3	实用脚本 16-3: 配置文件	443
16.3.1	用到的技术	443
16.3.2	概念	443
16.3.3	潜在的陷阱	443
16.3.4	脚本结构	443
16.3.5	脚本代码	444
16.3.6	调用结果	445
16.3.7	小结	445
16.4	实用脚本 16-4: 锁	445
16.4.1	用到的技术	446
16.4.2	概念	446
16.4.3	潜在的陷阱	446
16.4.4	脚本结构	448
16.4.5	脚本代码	450
16.4.6	调用结果	452

16.4.7	小结	455
第 17 章	演示	457
17.1	实用脚本 17-1: 太空游戏	457
17.1.1	用到的技术	457
17.1.2	概念	457
17.1.3	潜在的陷阱	460
17.1.4	脚本结构	460
17.1.5	脚本代码	461
17.1.6	调用结果	466
17.1.7	小结	468
第 18 章	数据存储与检索	469
18.1	实用脚本 18-1: 分析 HTML	469
18.1.1	用到的技术	469
18.1.2	概念	469
18.1.3	潜在的陷阱	470
18.1.4	脚本结构	470
18.1.5	脚本代码	471
18.1.6	调用结果	472
18.1.7	小结	474
18.2	实用脚本 18-2: CSV 格式化	474
18.2.1	用到的技术	474
18.2.2	概念	475
18.2.3	潜在的陷阱	475
18.2.4	脚本结构	475
18.2.5	脚本代码	477
18.2.6	调用结果	479
18.2.7	小结	480
第 19 章	数值	481
19.1	实用脚本 19-1: 斐波那契数列	481
19.1.1	用到的技术	481
19.1.2	概念	482
19.1.3	潜在的陷阱	482
19.1.4	方法一的结构	483
19.1.5	方法一的脚本	483
19.1.6	方法一的调用结果	484
19.1.7	方法二的结构	484

19.1.8	方法二的脚本	485
19.1.9	方法二的调用结果	486
19.1.10	方法三的结构	487
19.1.11	方法三的脚本	488
19.1.12	方法三的调用结果	488
19.1.13	小结	490
19.2	实用脚本 19-2: PXE 启动	490
19.2.1	用到的技术	490
19.2.2	概念	490
19.2.3	潜在的陷阱	491
19.2.4	脚本结构	491
19.2.5	脚本代码	492
19.2.6	调用结果	495
19.2.7	小结	497
第 20 章	进程	499
20.1	实用脚本 20-1: 进程控制	499
20.1.1	用到的技术	499
20.1.2	概念	499
20.1.3	潜在的陷阱	501
20.1.4	脚本结构	501
20.1.5	脚本代码	503
20.1.6	调用结果	509
20.1.7	小结	514
第 21 章	国际化	515
21.1	实用脚本 21-1: 国际化	515
21.1.1	用到的技术	516
21.1.2	概念	516
21.1.3	潜在的陷阱	517
21.1.4	脚本结构	518
21.1.5	脚本代码	518
21.1.6	调用结果	522
21.1.7	小结	524

第IV部分 参考信息

附录 补充材料	527
术语表	531

第 I 部分

基 本 概 念

- 第 1 章 Unix、GNU 和 Linux 的历史
- 第 2 章 环境的搭建
- 第 3 章 变量
- 第 4 章 通配符扩展
- 第 5 章 条件执行
- 第 6 章 使用循环进行流控制
- 第 7 章 变量(续)
- 第 8 章 函数和库
- 第 9 章 数组
- 第 10 章 进程
- 第 11 章 shell 的选择与使用

第 1 章

Unix、GNU 和 Linux 的历史

Unix 有着悠久的历史，并且 Linux 就是由其发展而来的，因此要了解 Linux 就必须了解 Unix，而了解 Unix 就必须了解其历史。Unix 诞生之前，开发人员通常要提交一摞摞的打孔卡片，其中每张卡片代表一个命令或命令的一部分。计算机读取并顺序执行这些卡片。完成这些工作后，开发人员会得到计算机产生的输出。从提交到计算机完成输出通常需要几天的时间。如果代码中有错误，那么输出就会有错误，而且开发人员必须重新开始。后来，电传打字机以及各种形式的分时系统极大地加速了开发过程，但是计算模型基本上是一样的：一个字符序列(打孔卡片或键盘上的按键——仍然只是字符串)以批量作业的形式提交并运行(或者运行失败)，然后等待相应结果的输出。这对今天而言意义重大，因为这是所有计算系统中数据的传输方式——按顺序传输的字符序列。无论是文本文件、网页、电影或者音乐，它们全都只是 0、1 序列，并且一直如此。任何看起来有些不一样的东西都只是缘于 0、1 序列上层的接口。

20 世纪 60 年代中期，Unix 与其他各种交互式、分时系统开始起步。Unix 及其约定至今一直处于计算实践活动的中心地位；它的影响体现在了 DOS、Linux、Mac OS X 甚至 Microsoft Windows 上。

1.1 Unix

1965 年，贝尔实验室与通用电气公司(GE)加入了麻省理工学院(Massachusetts Institute of Technology, MIT)的一项名为 MULTICS(Multiplexed Information and Computing System, 多工信息与计算系统)的项目。Multics 一开始是要被设计成一个稳定的分时操作系统。“多工”的要求给项目带来了不必要的复杂度，最终导致贝尔实验室于 1969 年放弃该项目。Ken Thompson、Dennis Ritchie、Doug McIlroy 与 Joe Ossanna 保留了该项目的一些思想，大大降低了复杂度，并开发出了 Unix(该名称是对 MULTICS 的戏谑，因为 Unix 是受 MULTICS 启发而成的简化版的操作系统)。

Unix 的一个早期特征是引入了管道——这一机制由 Doug McIlroy 酝酿了几年，并由

Ken Thompson 在 Unix 中实现。此外，管道同样采用串行数据流的概念，但是它引入了让数据流通过的 `stdin` 和 `stdout`。之前就有过与管道类似的东西，并且概念都相当简单：由一个进程产生的输出作为另一个命令的输入。Unix 的管道方法引入的概念给系统其余部分的设计带来了显著的影响。

大多数命令都有一个文件作为参数，但是现有的命令被修改成默认从“标准输入”(stdin)读取，并向“标准输出”(stdout)输出；管道可以将这些数据从一个工具“导向”另一个工具。这是个新概念，它很大程度上定义了 Unix 的 shell；它使整个系统成为许多具有通用功能工具的集合，这与庞大的专用应用程序的做法是相违背的。这一思想被概括为“做一件事并把它做好”。在保持与 Unix 工具的兼容性的同时，GNU 工具链被编写出来用于代替 Unix。GNU 项目的开发人员经常要负责重写 Unix 工具并增加附加功能，同时还要恪守“做一件事并把它做好”的哲学。



GNU 项目由 Richard Stallman 于 1983 年发起，目的是为了使用自由软件代替具有专利权的商业 Unix。在 1991 年 Linux 内核项目启动之前，GNU 几乎完成了替换所有用户空间工具的任务。实际上，GNU 工具与它们原始的 Unix 版本相比，至少实现了相同的功能，并且出于实际经验经常会提供额外的有用特性。独立测试说明了 GNU 工具实际上可能比其传统的 Unix 版本更可靠(<http://www.gnu.org/software/reliability.html>)。

例如，`who` 命令列出登录到系统的用户，每一行代表一个登录会话。`wc` 命令对字符、单词与文本行进行计数。因此，下面的代码将显示有多少用户已登录：

```
who | wc -l
```

`who` 工具没有必要具备对登录用户数目进行计数的选项，因为通用的 `wc` 工具已经能够做到这一点。虽然对于 `who` 而言不会太省事，但如果用于整套工具(包括任何需要编写的新工具)，这将省去大量的精力并降低复杂度，而复杂度越高便意味着引入额外漏洞的可能性越高。当这样的方法用于更加复杂的工具时，如 `grep` 或者 `more`，系统的灵活性将随着每个增加的工具得到增强。



对于 `more` 命令而言，情况实际上比表面看来要复杂；首先它必须找出行数与列数。此外，系统中存在提供这一信息的工具集。按照这种方式，工具链中的每个工具都可以被其他工具使用。

同样地，这样的系统意味着用户不必学习每个实用程序实现其“单词计数”特性的方法。实际上有一些标准的开关：`-q` 一般表示 Quiet(静默)，`-v` 一般表示 Verbose(详细)等。但是，如果 `who -c` 表示“对列出的条目进行计数”，而 `cut -c <n>` 表示“剪切开头 n 个字符”，那么 `-c` 在这两个命令中的含义就不一致了。最好每个工具都完成各自的任務，并让 `wc` 为

它们完成计数的工作。

再举一个更加极端的例子，`sort` 工具只对文本进行排序。它可以按字母表顺序或数值顺序(二者的区别在于，按照字母表顺序 10 在 9 之前，但是按照数值顺序 10 在 9 之后)进行排序，但它并不对内容进行搜索，也不是每次显示一页。`grep` 和 `more` 可以通过管道与 `sort` 组合起来，实现如下面的管道程序所示：

```
grep foo /path/to/file | sort -n -k 3 | more
```

该管道程序将在 `/path/to/file` 中搜索 `foo`。命令的输出(`stdout`)将流入到 `sort` 命令的 `stdin`。想象花园中的一根喷水软管，用它收集 `grep` 的输出，并连接到 `sort` 的输入。`sort` 从 `grep` 接收到过滤过的序列，并将排序后的结果输出到 `more` 的 `stdin`，由 `more` 来读取过滤与排序过的数据，并对其进行分页输出。

我们最好理解其中的实际过程：该过程与人们直观认为的相反。首先运行的是 `more`，它的输入被连接至一个管道。然后是 `sort`，并且其输出被连接至之前那个管道。接着创建第二个管道，并且将 `sort` 的 `stdin` 连接至该管道。最后执行 `grep`，将其 `stdout` 连接至与 `sort` 进程相连的那个管道。

当 `grep` 开始运行并输出数据时，数据顺着管道流向 `sort`，`sort` 将其输入进行排序并顺着管道输出到 `more`，由 `more` 对管道输出的全部内容进行分页输出。在有错误的情况下，这样的过程会出现不同的行为。如果 `more` 这个词键入错误，则不会有任何事情发生。如果 `grep` 这个词键入错误，则 `more` 和 `sort` 会执行到错误被检测出来。对于本例而言，这样的错误没有什么影响。但是，如果处于管道程序下游的命令具有某种永久性的效果(例如创建或修改一个文件)，那么即使没有执行整个管道，系统的状态还是会改变。

1.1.1 “一切皆文件”与管道

Unix 中还引入了一些关键性的概念。其中之一便是著名的“一切皆文件”的设计方法。按照这种设计思路，设备驱动程序、目录、系统配置、内核参数以及进程全都表示为文件系统上的文件。任何事物，无论是纯文本文件(例如 `/etc/hosts`)、块设备或字符设备驱动程序(例如 `/dev/sda`)或者是内核状态与配置(例如 `/proc/cpuinfo`)，它们都用文件的形式表示。

管道的存在使得系统中的工具在编写时假设它们处理的是文本流，而且实际上大多数系统配置也是以文本的形式存在。可以使用现有的工具对配置文件进行排序、搜索、重新格式化甚至比较与重组操作。

“一切皆文件”的概念与 4 种可以执行的文件操作(`open`、`close`、`read`、`write`)意味着 Unix 实际上使用了一种简洁明了的系统设计方法。`shell` 脚本本身也是文本形式的系统实用程序。我们可以编写下面这样的脚本程序：

```
#!/bin/sh
cat $0
echo "===="
tac $0
```

这段代码使用 `cat` 工具简单地输出一个文件的内容，然后使用 `tac` 工具将内容按相反的顺序输出(`tac` 的用途可以顾名思义，因为字母 `cat` 颠倒过来就是 `tac`，这也是典型的 Unix 式

的幽默)。变量\$0是一个由系统定义的特殊变量，它包含所谓当前运行程序的名称。

因此该命令的输出如下所示：

```
#!/bin/sh
cat $0
echo "==="
tac $0
===
tac $0
echo "==="
cat $0
#!/bin/sh
```

前4行是cat的结果，第5行是echo语句的结果，而最后4行则是tac的输出。

1.1.2 BSD

AT&T公司的贝尔实验室不能出售 Unix，因为它当时处于电信垄断地位，这样便阻碍了 Unix 进入诸如计算等其他行业。因此，AT&T 开始分发 Unix，特别针对的是渴望免费获取操作系统的大学。实际上，高校能获取到源代码有额外的好处，尤其是对于系统管理员，也包括学生。用户与管理员不仅可以运行操作系统，还能够查看(与修改)使操作系统运行的代码。对于 AT&T 而言，提供源代码的访问权是很容易办到的；公司(在当时那个阶段)也不是特别愿意独自开发和为 Unix 提供支持，这样一来用户可以自行提供支持。最终的结果是许多有着 Unix 经验的大学生进入了各行各业，当工作需要操作系统时，他们便会建议使用 Unix。Unix 使用量的增长是因为它在喜爱其简明设计的用户之间很流行，也是缘于其偶然的发布方式。

尽管 Unix 以免费或低价方式分发，并且包含了源代码，但根据自由软件基金会(Free Software Foundation, FSF)的定义，Unix 不是自由软件。FSF 对自由软件的定义是关于自由而不是价格。Unix 许可证禁止 Unix 用户向其他人员进行重新发布，尽管许多用户开发了自己的补丁与一些使用 Unix 子许可证的共享补丁(对于没有获得 AT&T 的 Unix 许可证的用户而言，这些补丁可能没有用处。核心软件仍然是 Unix；任何补丁都只是对它的修改)。加州大学伯克利分校的伯克利软件套件(Berkeley Software Distribution, BSD)创建和发布了许多这样的补丁，从而修复了漏洞、增添了特性并改进了 Unix。“自由软件”与“开放源码”这两个术语不会存在很长的时间，但是所有软件的发布都是基于一个思想：如果某物有用，则它应当被共享。TCP/IP 作为 Internet 的两个核心协议通过 BSD 进入了 Unix，以同样方式进入 Unix 的还有 BIND、域名系统(Domain Name System, DNS)服务器和 Sendmail 邮件传输代理(mail transport agent, MTA)。最终，BSD 为 Unix 开发了足够多的补丁，以至于该项目实际上取代了全部的原始 Unix 源代码。经过一场诉讼案，AT&T 与 BSD 达成协议，BSD 中残留的 AT&T 部分的代码将被重写或以新的许可证发布，这样 BSD 的所有权将不属于 AT&T，并且可以自行发布。BSD 后来衍生出 NetBSD、OpenBSD、FreeBSD 以及其他一些变种。

1.2 GNU

我们已经提到 GNU 项目是从 1983 年开始的,而在当时大多数计算机制造商的软件都是以闭源的形式连同硬件一起发布的,该项目是针对闭源的一项抵抗运动。早期一直都有社区在用户之间共享源代码,如果有人认为可以进行改进,那么他们可以按照其意愿修改代码。这并没有以法律文书的形式确定下来,它只是开发人员之间自然形成的一种文化。如果有人对软件的某部分感兴趣,何不给他一份软件的副本(通常以源代码的形式分发,便于他人对其进行修改并运行在自己的系统上。当时在一台机器上编译的二进制软件很少能够在另一台机器上运行)?正如 Stallman 在 *Free Software, Free Society*(2002)一书的第 1 章中所说的那样,“软件的共享……与计算机一样古老,就像菜谱的共享与烹饪一样古老”。

从 20 世纪 70 年代到 80 年代早期,Stallman 与 MIT 的其他开发人员一直使用非兼容分时系统(Incompatible Timesharing System, ITS)。随着那一代硬件的衰落,新的硬件涌现出来,并且随着硬件行业的发展与新增的特性,新的机器都带有预先定制的操作系统。在当时,操作系统通常与硬件有着很强的相关性,因此随着运行 ITS 和 CTSS 的硬件被新的设计取代,这两种操作系统也逐渐衰亡。



ITS 是对当时同样在 MIT 开发的 IBM 的兼容分时系统(Compatible Time Sharing System, CTSS)的戏谑。CTSS 中的 C 强调它与 IBM 早期大型机的某些兼容性。而 ITS 中的 I 则标榜其叛逆的非兼容性。

Stallman 的转变是从他试图修复一个打印机驱动程序开始的。他希望打印机在阻塞(经常发生)时能够警告已提交任务的用户,让用户来解决阻塞问题。当时的打印机可以供每个人使用。如果用户提交的任务将打印机阻塞,则要等到问题解决才能得到打印输出。但是在这之后已经提交任务的用户则必须等待更长的时间。MIT 的用户在提交打印任务时,经常遇到这样恼人的情况,等了几个小时(当时的打印机比现在要慢得多),最后发现打印机甚至在自己提交打印任务之前就已经阻塞了。于是 Stallman 希望修改其中的代码。他并没有指望原来的开发人员能为他开发这一特性,而是愿意自己进行修改,于是向开发人员索要源代码的一份副本。他被拒绝了,因为驱动软件包含了与打印机工作原理有关的专利信息,而这些信息对于其他打印机制造商而言可能是很有价值的商业竞争信息。

触动 Stallman 的不是软件特性本身,而是开发人员不愿意与其他开发人员共享代码的现实。Stallman 不能认同这样的态度,因为直到当时他都认为代码的共享是理所当然的。问题在于软件——尤其是那个打印机驱动程序——不像 Stallman 曾使用过的以前的操作系统一样免费(这里没有自由的含义)。这样的问题充斥了整个行业,而且不只针对某一个平台,所以更换硬件也无济于事。



GNU 是 GNU's Not Unix 的递归缩写。如果将缩写 IBM 扩展开, 则得到 International Business Machines, 并且往下就不能展开了。如果将 GNU's Not Unix 扩展开, 则得到 GNU's Not Unix's Not Unix。再扩展, 则得到 GNU's Not Unix's Not Unix's Not Unix, 依此类推。这是一种典型的“黑客式幽默”(耍点小聪明或者标新立异), 它通常是一种相当冷的幽默。在 `grep` 手册页的末尾, NOTES 部分下的一行注释——GNU's not Unix, but Unix is a beast; its plural form is Unixen(GNU 不是 Unix, 但 Unix 是野兽, 其复数形式为 Unixen)——则是对 Unix 的戏谑。

Richard Stallman 是一个意志坚定、思想很有逻辑性的人(他将自己描述为“边缘性孤独症者”), 并且决心使用他所知的唯一方式来解决: 编写新的操作系统来维护早期那种不成文的自由, 允许对于系统有同等的访问权, 包括使系统运行的代码。由于当时没有这样的操作系统, 他必须自己编写, 而且他确实这么做了。

Stallman 打响了抗争的第一枪!

From CSvax:pur-ee:inxcl:ixn5cl:ihnp4!houxm!mhuxi!eagle!mit-vax!miteddie!

RMS@MIT-OZ

Newsgroups: net.unix-wizards,net.usoft

Organization: MIT AI Lab, Cambridge, MA

From: RMS%MIT-OZ@mit-eddie

Subject: new Unix implementation

Date: Tue, 27-Sep-83 12:35:59 EST

自由的 Unix!

从这个感恩节开始, 我将要编写一个名为 GNU 的(表示 Gnu's Not Unix)与 Unix 完全兼容的软件系统, 并免费分发给能够使用的人。这需要很多的时间、财力、程序与设备的投入。

首先, GNU 需要一个内核以及所有用于编写与运行 C 程序的实用工具: 编辑器、shell、C 编译器、链接器、汇编器与其他一些工具。然后我们将增加一个文本格式化程序、一个 YACC、一个帝国游戏、一个电子表格和数以百计的其他程序。最终我们希望能包含 Unix 系统中带有的所有有用工具, 以及其他任何有用的东西, 包括联机与硬拷贝文档。

GNU 将能够运行 Unix 程序, 但与 Unix 有区别。我们将基于其他操作系统的经验进行改进, 使得 GNU 更加方便。特别是, 我们计划使用更长的文件名、文件版本号、防崩溃文件系统、文件名补全与独立于终端的显示支持, 最后是基于 Lisp 的视窗系统。Lisp 程序与普通的 Unix 程序可以通过该视窗系统来共享屏幕。C 语言与 Lisp 语言可以作为系统编程语言使用。我们还需要优于 UUCP 且基于 MIT 的 chaosnet 协议的网络软件。另外, 可能还需要某些与 UUCP 兼容的软件。

关于我本人

我叫 Richard Stallman, 原始 EMACS 编辑器的发明人, 现在就职于 MIT 的人工智能实验室。我在编译器、编辑器、调试器、命令解释器、非兼容分时系统与 Lisp Machine 操作系统方面开展了广泛性的工作。我最先在 ITS 中实现了独立于终端的显示支持。另外, 我已经实现了一个防崩溃文件系统, 以及为 Lisp 机器实现了两个视窗系统。

编写 GNU 的目的

我认为的黄金准则是, 如果我喜欢一个程序, 那么我必须与喜欢它的其他人进行分享。从良知的角度, 我无法签署保密协议或者软件许可协议。

因此, 为了可以继续使用计算机而又不破坏自己的原则, 我决定将足够多的自由软件集合起来, 这样就可以不使用非自由软件。

贡献的方式

我正在向计算机制造商募集机器与资金, 以及向个人募捐程序与工时。

已经有一个计算机制造商答应提供一台机器, 但我们可能要使用更多的机器。如果您捐赠计算机, 可以预见到的一个结果是, GNU 不久将能运行在这些机器上。机器最好能在居民区操作, 而且不需要复杂的冷却或电源系统。

个人程序员的贡献方式可以是编写一些 Unix 程序的兼容副本, 然后将其交给我。对于大多数项目, 这样的兼职工作可能很难协调。独立编写的部分放在一起可能无法运行。但对于替换 Unix 这样的特殊任务, 这样的问题是不存在的。大多数接口规范已经由 Unix 的兼容性确定下来了。如果每个贡献的程序都能用于 Unix 的其余部分, 则很可能用于 GNU 的其余部分。

如果我募得资金, 我就能雇佣一些全职或兼职的员工。工资不会很高, 但我要寻找的是认为人道主义与金钱一样重要的那些人。这样一来, 就可以使这些人全身心地投入到 GNU 的工作中, 同时确保他们生活无虞。

如有疑问请联系我本人。

当时已经有了 Unix, 并且相当成熟, 模块化也很好。因此 GNU 项目开始的目标是使用自由软件代替 Unix 的用户空间工具。内核是总体目标的另一部分, 尽管操作系统不能只有一个独立的内核——内核需要编辑器、编译器和链接器来构建, 以及某种初始化进程来启动。因此, 现有的专利软件系统被用来汇集一个免费的足以用来对自身进行开发的系统, 并最终编译内核。这一主题一直都没有被忽略; 选用 Mach 微内核模型与最新的操作系统内核设计思想一致。HURD 内核在相当一段时间内都是可用的, 尽管它已经被某个新兴的内核超过, 而该新兴内核也是在 GNU 工具下开发的, 并可以与 GNU 工具一起使用。



HURD 表示 Hird of Unix-Replacing Daemons, 因为它的微内核方法使用多个用户空间后台进程(在 Unix 传统中称为守护进程)来实现 Unix 中的单内核的功能。而 HIRD 表示 Hurd of Interfaces Representing Depth。这又是一个像 GNU 一样的递归缩写, 但这次是一对相互递归的缩写。这也是在表示 Gnus 的集合名词 herd 上开的一个玩笑。

当对于自由软件不成文的解释失败后, Stallman 需要寻找一种全新的方法来确保可自由发布的软件保持自由的特性。GNU 通用公共许可证(General Public License, GPL)巧妙地达到了这一目的。GPL 使用版权确保许可证本身不能被修改;许可证的余下部分声明,发布对象具有代码的完全占有权的条件是,他必须将同样的权利授权给他的发布对象(无论修改与否),并且不能修改许可证。如此一来,所有的开发人员(和用户)都处于同一个平台。在这个平台上,代码实际属于所有参与进来的人们,并且没有人能对许可证进行修改。这样就确保了平等性。软件的某一部分的作者可能对软件使用双重许可证,如 GPL 和一个更具限制性的许可证,并且已经有很多这样的例子——如 MySQL 项目。

GNU 项目肩负的任务之一当然是编写作为自由软件的 shell 解释器。Brian Fox 编写了 bash(Bourne Again Shell) shell——该名称来源于 Steve Bourne 编写的原始/bin/sh, 其名为 Bourne shell。因为 bash 带有 Bourne shell 的特性,并且还添加了一些特性,所以它便成为了 Bourne Again Shell。Brian 还编写了 readline 工具,用于在提交文本输入行进行分析之前可以对它们进行灵活的编辑。或许是这一最有意义的特性使得 bash 成为主要的交互式 shell。Brian Fox 是自由软件基金会的第一位员工,该基金会是组织与协调 GNU 项目的实体。



此时您也许已经看出其中的模式; 尽管 bash 不是递归缩写,但它说明了基于 Bourne shell 的事实。同样,在 bourne again 中暗含了对于原始 Bourne shell 改进的含义。

1.3 Linux

芬兰的一名大学生 Linus Torvalds 当时正在使用 Minix, 这是由 Vrije 大学(位于阿姆斯特丹)的讲师 Andrew Tanenbaum 编写的一个简单的 Unix 克隆版本。但是 Torvalds 为其特性的匮乏感到沮丧,而且实际上 Minix 没有充分利用(相对而言仍然较新的)Intel 80386 处理器,尤其是它的“保护模式”可以更好地隔离内核与用户空间。很快他便有了可用的 shell,并随后使 GCC(GNU C compiler, 现在称为 GNU Compiler Collection, 因为它已经扩展到编译 C、Fortran、Java 与 Ada 语言)运行起来。在当时,内核加上 shell 和编译器足以“引导”系统启动——它们还能用来构建出自身的副本。

Torvalds 在新闻组上的发帖

1991 年 8 月 25 日, Torvalds 在 MINIX 新闻组 comp.os.minix 中发布了如下内容:

From: torvalds@klaava.helsinki.fi (Linus Benedict Torvalds)

To: Newsgroups: comp.os.minix

Subject: What would you like to see most in minix?

Summary: small poll for my new operating system

这里向使用 minix 的所有人问个好。

我正在为 386(486)AT 的克隆品编写一个(免费)的操作系统(只是一个嗜好,不会像 GNU 那样庞大且专业)。从 4 月份开始萌发这一想法,现在差不多开始着手了。我希望得到任何

喜爱或不喜爱 minix 的人的反馈意见，因为我的操作系统在其他方面与 minix 有些相似(由于实际原因，文件系统具有相同的物理层)。

最近我已经移植了 bash(1.08)和 gcc(1.40)，而且看起来能正常工作。这就表明我可以在几个月之内编写一些实用的东西了。我想知道大多数人都希望它能拥有哪些特性。

欢迎提出任何建议，但我不保证会去实现。

Linus Torvalds torvalds@kruuna.helsinki.fi

有趣的是，Torvalds 认为 GNU 项目的成功是必然的。经过 8 年的发展，GNU 基本实现了大部分目标(除了内核)。在起初犯了试图撰写自己的许可证(如果您对知识产权法的国际适用性的细节不熟悉的话，那么我们通常不建议这么做)的错误之后，Torvalds 也非常自然地使用 GNU GPL(v2 版本)作为 Linux 内核项目的许可证。

实际上，相比于 Linux 内核下的 shell 脚本编程，本书更多的是关于使用 Unix 和 GNU 工具进行 shell 脚本编程。总体而言，本书涉及的大部分工具都是来自自由软件基金会的 GNU 工具：grep、ls、find、less、sed 和 awk，当然还有 bash 本身，以及 diff、basename 和 dirname。Linux 下 shell 脚本编程使用的大多数关键命令都是 GNU 工具。因此，有些人倾向于用 GNU/Linux 来描述这种 GNU 用户空间与 Linux 内核的组合。本书的目的在于避免过分的政治狂热的同时保持技术上的准确性。RedHat Linux 是 RedHat 的 Linux 发行版，因此称之为 RedHat Linux。Debian GNU/Linux 则倾向于表现其含有 GNU 的成分，所以当特别谈及 Debian 时，我们也保持同样的态度。当谈及 Linux 内核时，我们就使用 Linux 这个词；而对于某个 GNU 工具，则会给出它的名称。渴望头版头条的新闻记者可能偶尔将社区中存在的分歧想象得比实际情况要大得多。就像所有的大家庭一样，社区内部也有矛盾——经常是激烈且公开的矛盾——但是本书不打算激化这些矛盾。



Unix 是被设计成由工程师来使用的。如果某人想用它来做某件事情，他或她必须准备好学习系统的工作原理以及操作方法。Unix 的整体设计优雅而简洁(“一切皆文件”、“做一件事并把它做好”等)，这意味着从系统的一部分学到的原理可以适用于系统的其他部分。

GNU/Linux 系统的流行，特别是在桌面 PC 以及便携系统(不仅仅是在阴暗的数据中心里面嗡嗡作响的服务器)上相对广泛的使用，孕育出了新一代基于共享哲学的工具集，并且抛开了任何不必要的历史背景。

Microsoft Windows 有着一套很不一样的哲学：最终用户不必关心底层系统的工作原理，因此也不应当知晓其原理，即使是有经验的专家，这都缘于 Windows 使用的是闭源许可证。两种哲学的区别不在于质，甚至也不在于量：Windows 采用的是一种不同的方法，它假定了一种层次关系，其中开发人员知晓软件的原理，用户无须知晓任何原理。

所以，许多有经验的 Windows 用户在收到一份 GNU/Linux 发行版后，发现如果要对系统进行某种“显然”应当要做的配置时，他们必须手动编辑文本文件或者指定某个特殊参数，这让他们感到很失望。这种灵活性其实是系统的优势而非劣势。在 Windows 模型中，

用户不必学习操作系统，因为他们不被允许对操作系统进行任何重要的配置：如内核调度器、文件系统以及窗口管理器。这些配置已被开发人员调整到“一体适用”的程度来满足大多数用户的需求。

1.4 本章小结

尽管无需 GNU/Linux 的历史背景也可以为其管理与编写 shell 脚本，但是如果对于它们的来历没有一定的理解，很多看上去很奇怪的地方就会显得毫无道理可言。为 RedHat、SuSE 或 Ubuntu 这样典型的 Linux 发行版与为嵌入式设备编写脚本存在差异，因为后者很可能没有整套的 GNU 工具集而是运行 busybox。为商业 Unix 编写脚本也有一些区别，就像 Web 开发人员要考虑确保网站在多种平台的多个浏览器上显示，编写稳定的跨平台 shell 脚本需要一定量的测试。

甚至在为典型的 Linux 发行版编写脚本时，对于背景的了解也是有帮助的。`/etc/sysconfig` 是否存在？初始化脚本是位于 `/etc/rc.d/init.d` 还是 `/etc/init.d` 中，或者是否存在？从某个特定发行版的哪些特性能看出该发行版沿袭了什么样的传统？对于系统历史背景的了解有助于理解是使用 `tar xzf` 还是 `tar -xzf`，使用 `/etc/fstab` 还是 `/etc/vfstab`，以及运行 `killall httpd` 是只关闭 Apache 进程(GNU/Linux 的做法)还是关闭整个系统(Solaris 的做法)！

从这种历史的多元性出发，第 2 章比较了 Unix 与 GNU/Linux 环境之间选择的多样性。

环境的搭建

在运行与测试本书代码之前，我们需要搭建类似 Unix 的环境。虽然阅读本书的读者可能已经能够使用 Unix 或 Linux 系统，但是本章将介绍一些操作系统环境与编辑器，以及它们的获取途径与使用方法。也可以考虑使用虚拟机或者至少在已有系统上创建一个独立的账户来运行本书代码。

尽管 GNU/Linux 与 bash shell 可能是当前最流行的操作系统与 shell 组合，并且本书主要使用这种组合，但还有很多其他的操作系统与 shell 可供选择。对于 shell 脚本编程，不同的操作系统一般而言不会有太大区别，所以本章更多的是介绍一些操作系统与编辑器。

2.1 操作系统

首先值得一提的是，Linux 不是唯一的选择。还有其他一些免费的操作系统可供选择，包括 BSD(FreeBSD、NetBSD 和 OpenBSD)、Solaris Express、Nexenta 等。然而，有很多 GNU/Linux 发行版，而且一般都提供对大量硬件和软件的良好支持。它们中的大多数都可以下载得到，并且可以完全合法地使用，甚至是用于产品开发。另外，RedHat Enterprise Linux(RHEL)与 SuSE Linux Enterprise Server(SLES)提供受限的使用权与系统更新；Oracle Solaris 对于产品开发则有 90 天的试用期限限制。

2.1.1 GNU/Linux

RHEL 是基于 Fedora 的商业 Linux 发行版，在北美以及欧洲的大部分地区特别流行。因为 RHEL 光盘中包含 RedHat 商标与一些非自由软件(如 RedHat Cluster)，所以光盘的发布仅限于授权用户。然而，CentOS 项目从源代码重新编译了 RHEL，并且去掉了 RedHat 商标，这样便形成了一个在二进制与源代码上都与 RHEL 完全兼容的 Linux 发行版。这很有用，因为很多 Linux 商业软件都只有针对 RHEL 的测试与支持，但是这些软件供应商通常也要提供 CentOS 上的应用支持，即使他们不对操作系统本身提供支持。

RHEL 本身只能通过订购来获取。然而，CentOS 与 Oracle Enterprise Linux 是两个克隆

版本，它们从源代码中去掉了 RedHat 商标，并按照与 RedHat 二进制文件相同的编译方式进行了重新编译。CentOS 可以从 <http://centos.org/> 上获取，Oracle Enterprise Linux 可以从 <http://edelivery.oracle.com/linux> 上获取。

Fedora 是由社区维护并向 RHEL 贡献代码的发行版。该发行版有着高度活跃且总体而言非常技术化的用户群，并且大多数开发首先在 Fedora 上进行测试，然后再向上游项目提交(提交到的相关项目可能是 GNOME、KDE 与 Linux 内核等)。和 Ubuntu 一样，Fedora 每 6 个月发布一次发行版，但是支持周期只有短短一年。在 Fedora 中得到验证的技术被放到 RedHat Enterprise Linux 中。同样，Fedora 提供 KDE、XFCE 与 LXDE 桌面环境，以及主流的基于 GNOME 的桌面环境。Fedora 的 DVD 镜像可以从 <http://fedoraproject.org/> 上获取。

SLES 是 Novell 公司的企业版 Linux。它基于社区版的 OpenSUSE，SLES 与 OpenSUSE 在欧洲特别流行，部分原因在于 SuSE 源自 Novell 于 2004 年收购的一家德国公司。SuSE 与其他 Linux 发行版最大的区别在于它使用 YaST2 配置工具。SLES 有着相当稳定的发布周期：每 2~3 年都会发布一个新的主要发行版，所以它的更新频率比 RHEL 要高，而比大多数其他 Linux 发行版要低得多。

SLES 的评估版可以从 <http://www.novell.com/products/server/> 上获取。与 RedHat Enterprise Linux 一样，如果要使用完整版本则需要签订一份支持合同。

OpenSUSE 与 SLES 的关系就如同 Fedora 与 RHEL 的关系。OpenSUSE 可能不如 SLES 稳定，但是更集中于社区，并且技术更加前沿。这就好比测试版在正式版之前发布一样。OpenSUSE 可以从 <http://software.opensuse.org/> 上获取。它的主页是 <http://www.opensuse.org/>。

Ubuntu 基于 Debian 的“测试”分支，它在 Debian 的基础上增添了一些新特性与自定义功能。Ubuntu 的安装与配置很容易。Internet 上有很多提供 Ubuntu 支持的论坛。Ubuntu 可以说是美化过的 GNU/Linux 发行版。它每 2 年提供一次长期支持版(Long-Term Support, LTS)，对于桌面系统是 2 年，对于服务器系统是 5 年。另外还有每 6 个月发行一次的常规发行版。常规发行版按照 YY-MM 的形式进行编号，因此 10-10 版本(发行代号为 Lucid Lynx)是在 2010 年的 10 月发布的。尽管 Ubuntu 桌面系统已广为人知，但是其没有图形界面的服务器版的用户占有率还有待提高。

Ubuntu 的安装方法有很多——可以使用 CD/DVD 或 U 盘安装，甚至还可以在已有的 Windows 系统中安装。安装指南与可免费下载的光盘镜像以及种子文件可以从 <http://ubuntu.com/> 上获取。还有很多经过重新编译的 Ubuntu 系统：使用 KDE 而不是 GNOME 桌面环境的 Kubuntu、使用 XFCE 窗口管理器的 Xubuntu，以及包含教育软件的 Edubuntu 和经过剪裁用于上网本的 Netbook Edition。

Debian 在主流的 GNU/Linux 中属于较早的一个发行版。它拥有超过 1 000 人的开发团队，提供的软件包数量超过 30 000。尽管 Debian 项目计划加快稳定版本的发布频率，但是一般大约 5 年发布一次，所以当前的稳定版本在时间上可能已经相当久远。测试分支版本在很多用户中很流行，它提供最新的软件包，且没有不稳定版本具有的系统不可预测性。Debian 的 CD/DVD 镜像可以从 www.debian.org/CD/ 上直接下载，或者通过 BitTorrent 下载。

全世界有数以百计的 GNU/Linux 发行版可供使用。网站 <http://distrowatch.com/> 提供了极好的关于现有的发行版以及其他 Unix 与类似 Unix 的软件的信息。还有一些值得注意的发行版，它们是 Gentoo、Damn Small Linux、Knoppix、Slackware 和 Mandriva。

2.1.2 BSD

伯克利软件套件(BSD)是最古老的 Unix 分支之一。它已经发展分裂出很多不同的分支,这里列出其主要的 3 个分支。BSD 的每个分支都有不同的侧重点,这决定了它们的开发风格。

FreeBSD 可能是最容易获取的 BSD 分支,并且提供了相比其他分支更多的硬件支持。OpenBSD 是 NetBSD 的分支,一般被认为是现有的最安全的 Unix 系统。尽管它的开发进度通常很慢,但是开发出来的系统极其稳定与安全。OpenBSD 被广泛用于路由器或防火墙。对于 2011 年 5 月发布的 4.9 版本的 OpenBSD 而言,目前为止在默认安装中只发现了 2 个可利用的远程安全漏洞。有些操作系统在一个月之内就能发现很多安全漏洞。

NetBSD 是最具可移植性的 BSD 分支。它可以运行在 PC、Alpha 和 PowerPC 架构上,以及 ARM、HPPA、SPARC/SPARC64、Vax 和其他很多架构上。

2.1.3 商业 Unix

Oracle Solaris 的历史可以追溯到 1983 年,并且毫无疑问是现今市场上特性最丰富、开发活动最活跃的企业级操作系统。SunOS 原本基于 BSD,但随着向 Solaris 的迁移开始向 System V 风格的 Unix 转变。现在的 Solaris 系统带有原始的 Bourne shell(/bin/sh),以及 ksh93、bash、csh、tsh、tcsh 和 zsh 等 shell。Solaris 还可用于 SPARC 和 x86 架构。

Oracle Solaris 可以从 <http://www.oracle.com/technetwork/server-storage/solaris/downloads/index.html> 上获取,能免费用于非商业用途或者提供 90 天的商业用途试用期。Solaris Express 是开发中的 Solaris 的技术预览版。OpenSolaris 的分支 OpenIndiana 可以从 <http://openindiana.org/> 上获取。另一个分支 Nexenta 的用户空间使用 GNU 软件,可以从 <http://nexenta.org/> 上获取。

IBM AIX 是 IBM 开发的用于 Power 架构的 Unix,它基于 System V Unix,有精简版(有最多 4 个 CPU 核与 8GB 内存的限制)、标准版(不限制其可扩展性)以及企业版(添加了额外的监控工具和特性)。在编写本书时,AIX 的最新版本是 2010 年 9 月发布的 7.1 版本。

HP-UX 是 HP 开发的 Unix,它基于 System V Unix,可以运行在 PA-RISC 和 Intel Itanium 系统上。在编写本书时,HP-UX 的最新版本是 2008 年 8 月发布的 11iv3 版本。

2.1.4 Microsoft Windows

cygwin 环境运行在 Microsoft Windows 下,它提供了相当全面的 GNU 工具集。如果无法使用运行原生 shell 的操作系统,cygwin 则可以在不退出 Windows 系统的前提下提供完整的 bash shell 功能以及核心工具(ls、dd 和 cat——几乎是 GNU/Linux 发行版中包含的一切)。这意味着可以使用如 grep、sed、awk 以及 sort 这样在 Linux 下运行的 GNU 工具。要注意的是,cygwin 并不是模拟器——它提供了一个 Windows DLL(cygwin1.dll)与一套编译成 Microsoft Windows 可执行文件(.exe)的实用工具(主要是 GNU 工具)。它们并非模拟,而是原生地运行于 Windows 下。运用中的 cygwin 如图 2-1 所示。注意,有些二进制文件是以 Microsoft DOS 和 Windows 下使用的.exe 扩展名命名。

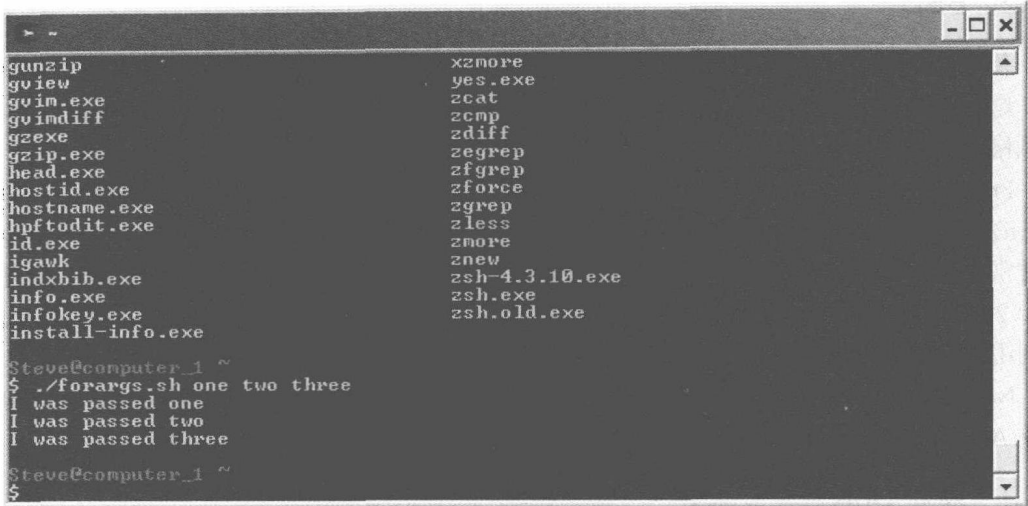


图 2-1

cygwin 可以从 <http://www.cygwin.com/> 上获取。

2.2 编辑器

以上提到的大多数操作系统中都有各种各样的文本编辑器可供使用。像 OpenOffice.org、Abiword 或 Microsoft Word 这样的字处理软件特别不适合编写程序，因为这些程序经常对文本进行修改，如拼写检查、大写字母以及格式化等。这样会导致脚本运行的不确定性。最好使用纯文本编辑器，也就是这里将介绍的。纯文本编辑器不对实际文件进行额外格式化，但这并不意味着它们就缺乏强大的功能；大多数文本编辑器都提供语法高亮功能，并且其中很多都为编辑 shell 脚本与其他文本文件提供更有用的特性。

2.2.1 图形化文本编辑器

对于图形界面环境，基于 GUI 的编辑器可能更易于使用。但学会如何使用非图形化编辑器也很重要，尤其是在无法使用 GUI 的情况下(X Window 系统配置损坏、通过远程 ssh 连接到服务器，以及通过串行接口连接到服务器等)。然而，对于日常使用而言，图形化编辑器的便利性还是吸引了一批用户。

1. gedit

gedit 是 GNOME 的默认文本编辑器，通常可以通过 Applications | Accessories | gedit Text Editor 找到。gedit 提供了基本的语法高亮功能，可以用来检查脚本中的语法错误。它还具有选项卡式窗口，并支持不同的文本文件格式(Windows、Linux 与 Mac OS 换行符，以及各系统下的字符编码)。运用中的 gedit 如图 2-2 所示。

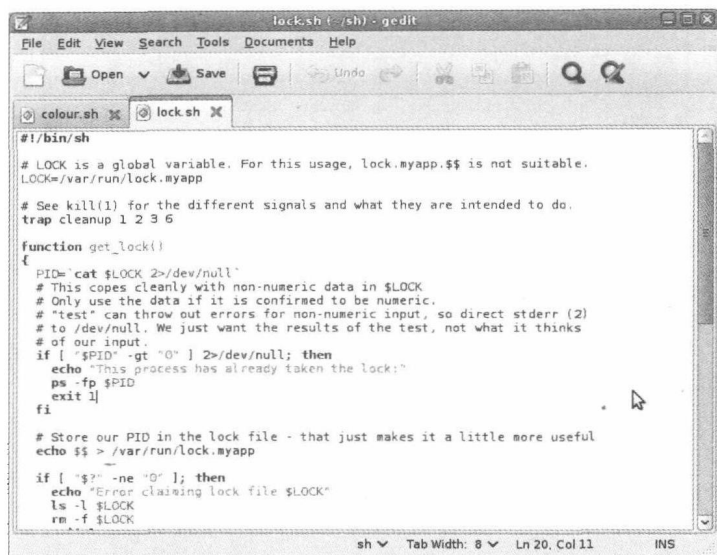


图 2-2

2. kate

kate 是 KDE 的默认文本编辑器。它提供了语法高亮、多重选项卡等功能，但它的特点在于其窗口形式的 shell，便于完全处在 kate 环境下编辑和运行脚本。运用中的 kate 如图 2-3 所示，其中的命令窗口运行了正在编辑的 shell 脚本。

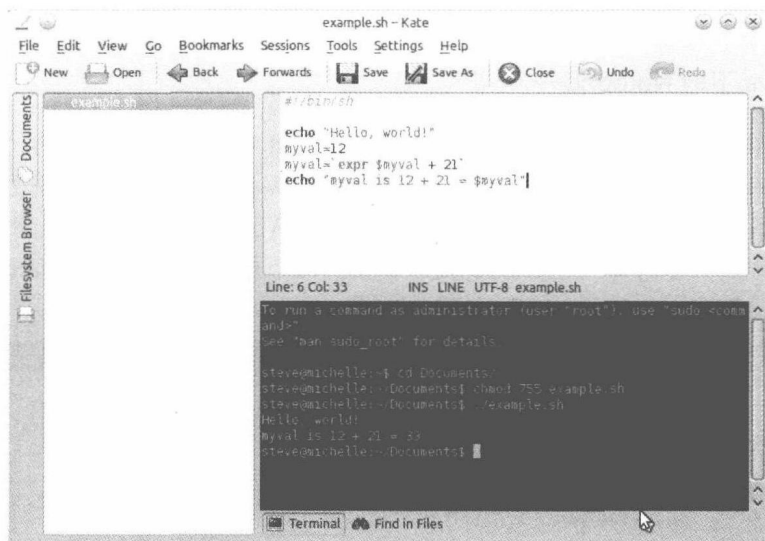


图 2-3

尽管 kwrite 更多用于编写简短的文档而非编写代码，但它也是 KDE 的一部分。

能够替换命令行工具 vi(大多数 Linux 发行版中都提供的是 VIM[Vi IMproved])的图形界面程序是 gvim。它具有双重特征，既有图形化界面(外观上与 gedit 几乎一样)，同时还保留了 vi 与 vim 常用的快捷键。运用中的 gvim 如图 2-4 所示，其中包含了两个正在编辑两个不同脚本的选项卡。

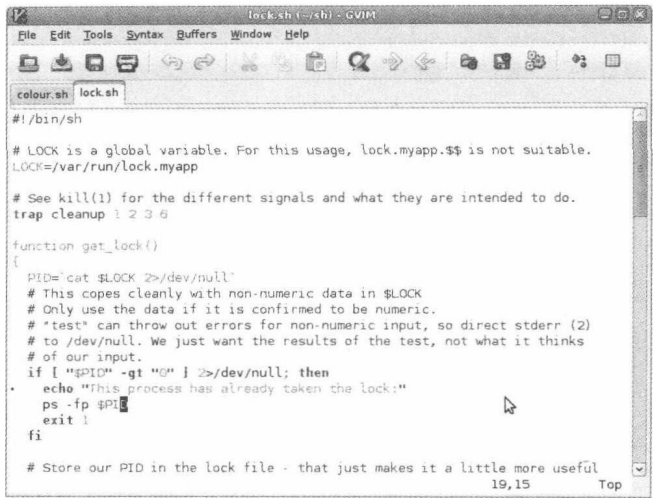


图 2-4

vim(既有命令行形式的 vim 也有图形化的 gvim)也可以在 Microsoft Windows 下使用，可以从 <http://www.vim.org/download.php#pc> 上获取。

3. Eclipse

Eclipse 是 IBM 开发的一个完整的集成开发环境(Integrated Development Environment, IDE)。它是用 Java 编写，并且本来是用于 Java 开发的，但也可以用于编写 shell 脚本。然而对于大多数 shell 编程任务而言，Eclipse 实在是大材小用。

4. Windows 下的 Notepad++

Notepad++(<http://notepad-plus-plus.org/>)是 Microsoft Windows 环境下的一款强大的编辑器，它按照 GPL(自由软件)许可证发布。它提供多种编程语言的语法高亮功能、强大的搜索选项以及很多使用插件架构实现的附加特性。作为 Windows 环境下的一款轻量级且功能完全的文本编辑器，Notepad++非常受欢迎。Windows 下原生视窗界面的 Notepad++如图 2-5 所示。

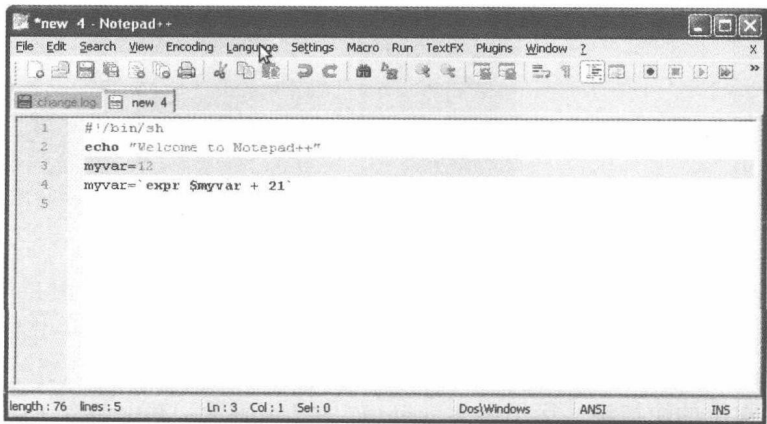


图 2-5

2.2.2 终端模拟器

对于终端模拟器, GNOME 下有 `gnome-terminal`, KDE 下有 `konsole`。XFCE 下的终端模拟器就叫 `Terminal`, 目的是在不依赖 GNOME 的前提下能够很好地替代 `gnome-terminal`。也有 `xterm`、`rxvt` 等其他终端模拟器。另外, 还有一个原生的 linux 终端模拟器, 就是登录到没有图形会话的 Linux 系统时所见到的模拟器。

`gnome-terminal` 是 GNOME 环境下的默认终端。它使用配置文件, 因此我们可以定义不同的外观设置。它还使用选项卡, 选项卡之间位置可以相互交换, 也可以从原窗口中移除选项卡。

`konsole` 是 KDE 环境下默认且非常灵活的终端模拟器, 在 `System` 菜单下可以找到。`konsole` 有一些特别有用的功能, 例如在终端无响应超过 10 秒(如当某个运行时间较长的作业向终端写完数据)或者从无响应状态中恢复(如当作业结束无响应状态并开始向终端写数据)时, 它会从 KDE 中弹出警告。

`konsole` 另一个优秀的特性是可以通过配置文件设置来定义双击时选中的“单词”。如果需要在双击时选中整个电子邮件地址, 就要将@符号与句点(.)置于对“单词”进行定义的列表中; 如果希望双击\$100 时选中的只是数字, 则不要将\$置于列表中。

如果需要在一些系统中运行相同的命令, 可以在不同的选项卡中登录每个服务器, 然后在当前窗口中选择 `Edit | Copy Input To | All tabs`。完成后要记得取消选定。

图形会话的原始终端模拟器是 `xterm`。尽管它不再常用, 但还是有必要熟悉一下它的使用方法, 因为偶尔也会遇到没有完整的图形环境可用的情况。

当登录到没有图形环境的 Linux 系统或者按下 `Ctrl+Alt+F1` 组合键时, 看到的的就是原生的 Linux 终端模拟器。它是基本的终端模拟器, 是 Linux 实际操作系统的一部分。它支持文本彩色显示、高亮与闪烁功能。

2.2.3 非图形化文本编辑器

还有很多基于命令行的文本编辑器, 它们有着各自的优势。

`vi` 是至今为止在系统管理员中使用最广的文本编辑器——要学习使用它刚开始比较困难, 因为它有两种操作模式(一是可以像普通编辑器一样输入文本的插入模式, 二是按键被转换成对文本进行操作的指令的命令模式), 并且不好区分在某个时刻所处的模式。关于模式真正需要了解的是, 按下 `Escape` 进入命令模式, 按下 `i` 进入插入模式。因为按下 `Escape` 总能进入命令模式, 所以 `Escape+i` 总能进入插入模式。一旦熟悉了这些, 并且学会 `vi` 的很多强大的常用命令后, 其他的编辑器相比之下就会显得缓慢而笨拙。尽管 `vi` 是 Unix 的一部分, 但大多数 GNU/Linux 发行版都包含了 `vim`(Vi Improved), 并且将 `vi` 作为 `vim` 的别名。`vim` 在保持 `vi` 兼容性的同时还添加了额外的功能。`vim` 带有一个 `vimtutor` 脚本, 它是一份 `vim` 教程, 其中包含了很多例子。`vimtutor` 教程的第 1 页如图 2-6 所示。

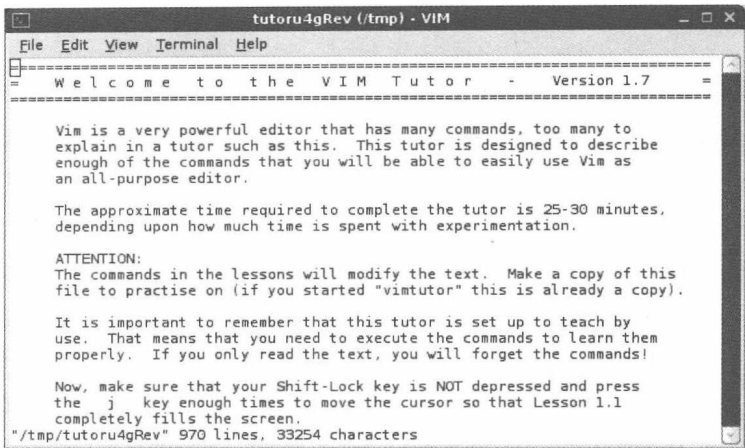


图 2-6

emacs 是另一个非常流行且带有大量插件的文本编辑器。如果将 **emacs** 完全配置好，甚至都可以不必使用 **shell**！它被形容成“热核字处理器”。与 **vim** 一样，**emacs** 刚开始是一个控制台下的非图形化文本编辑器，但如今也已经有了图形化版本。**emacs** 从一开始就考虑了跨平台特性，不对键盘上可用的按键进行任何假设，所以 PC 上的 **Ctrl** 键在 **emacs** 中称为 **Control** 键，**Alt** 键被称为 **Meta** 键。它们分别写成 **C-**与 **M-**，**C-f**(按下 **Control** 键后再按下 **f** 键)将光标向前移动一个字符，而 **M-f**(按下 **Alt** 键后再按下 **f** 键)则将光标向前移动一个单词。**C-x C-s** 用于保存，**C-x C-c** 用于退出。

vi 与 **emacs** 之间的对抗关系虽然历时长久，但总体而言还是比较温和的。只要没有谁被强迫使用“另一个”编辑器，**vi** 与 **emacs** 用户总能求同存异。图 2-7 显示了一个在 KDE 桌面环境下运行的图形化 **emacs** 会话。

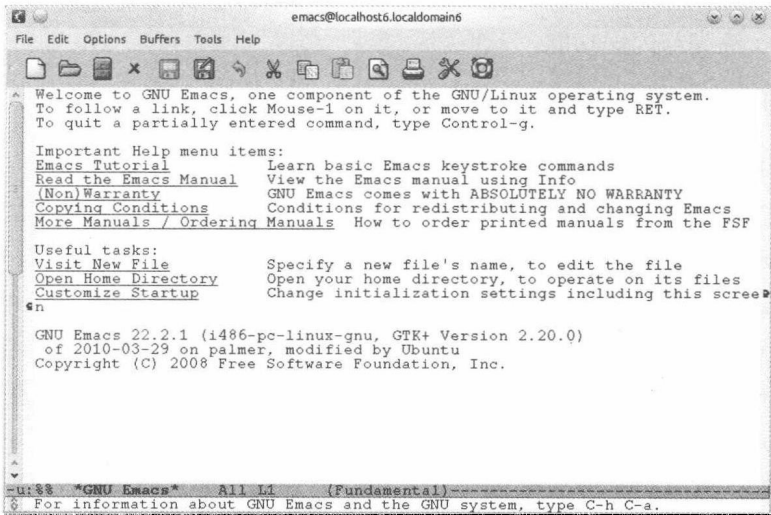


图 2-7

pico 与 **nano** 是更容易获取的文本编辑器。**pico** 是从华盛顿大学的 **pine** 邮件客户端的

编辑器发展而来的；nano 是 pico 的 GNU 克隆版本，并且是 Ubuntu 论坛推荐使用的编辑器。与 emacs 相似之处在于，很多命令都使用 Control 键(如 Ctrl-X 表示退出)；不同之处在于，屏幕下方总是显示一个与上下文相关的菜单，可以让特定上下文中可供使用的命令一目了然。图 2-8 显示的是正在使用 nano 编辑/etc/hosts 文件。

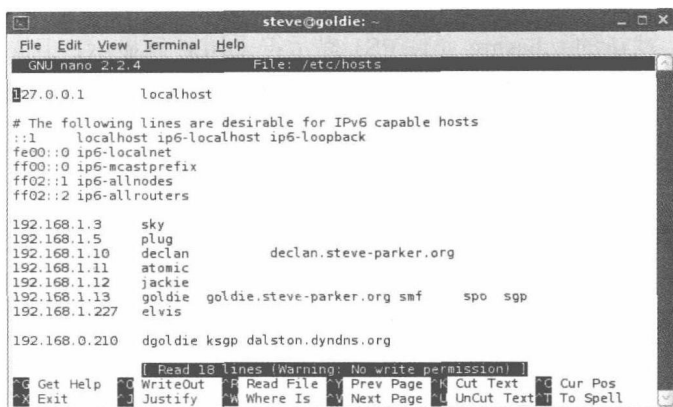


图 2-8

2.3 系统环境的搭建

Unix 与 Linux 是可定制性非常强的操作系统。用户可以使用多种方法将系统环境(定义 shell 运行方式的变量和设置)配置成喜欢的样子。如果发现有某个设置经常需要配置或者修改，通常都可以让系统来自动完成。下面介绍一些最有用的环境配置方法。

2.3.1 shell 配置文件

个人配置主要都集中在 ~/.profile(\$HOME/.profile)文件中。打开新的交互式 shell 时，配置文件的执行顺序是/etc/profile、/etc/bash.bashrc(如果使用 bash shell)、~/.profile，最后是 ~/.bashrc。所有的 shell 都会读取 ~/.profile，因此最好将通用设置写入其中，然后将与 bash 相关的设置写入 ~/.bashrc 中。该文件中可以设置变量与别名，如果有需要甚至可以运行命令。因为这些本地(用户特定的)配置文件都处于主目录下，并且以句点(.)开头，所以常规 ls 命令无法显示它们，因此它们被称为“点文件”。网络上有很多点文件的例子。<http://dotfiles.org> 是一个不错的点文件资源库。

环境变量

改变系统运行方式的环境变量有很多，可以交互式地对它们进行设置，但最好在 ~/.bashrc 文件中设置。

PS1 提示符

PS1 是基本的 shell 提示符，可以对它进行自定义。它在 bash 中的默认值为 \s-\v\\$，或者说是“shell 名称-版本号-美元符”——如 bash-4.1\$。还有很多提示符设置——详见 bash

手册页的 Prompting 部分。PS1 的常用值为 `\u@\h:\w$`——显示登录名、服务器名称和当前工作目录。例如：

```
steve@goldie:/var/log$
```

它表示作为用户 `steve` 登录到服务器 `goldie`，且当前处在 `/var/log` 目录下。

在 Debian 中，默认的 `~/.bashrc` 允许 PS1 提示符采用彩色显示，但同时说明了“对于终端窗口，注意力应当集中在命令的输出而不是提示符上”。如果真的需要彩色的提示符，可以将 `.bashrc` 文件中的 `force_color_prompt=yes` 一行解注释。

PATH

可以通过设置 PATH 环境变量来告诉 shell 查找要运行的程序(和脚本)的路径。系统主要的命令都在 `/bin/`、`/usr/bin/`、`/sbin/` 和 `/usr/sbin/` 中，但也可以在 `$HOME/bin/`、`$HOME/scripts/`、`/usr/local/bin/` 或其他位置放置用户编写的脚本。将这些路径添加到 PATH 中，这样可以 shell 找到这些路径，即使当前并不处于这些路径下：

```
PATH=${PATH}:${HOME}/bin
```

如果没有 PATH，则需要提供命令的显式路径(绝对路径或者相对路径)。例如：

```
$ myscript.sh
bash: myscript.sh: command not found
$ /home/steve/bin/myscript.sh
... or:
$ cd /home/steve/bin
$ ./myscript.sh
```

从安全的角度考虑，将点号(.)添加到 PATH 中这一做法是非常糟糕的，尤其是将其添加到 PATH 的开头。如果进入某个目录并运行某个命令(可能是 `ls`)，则真正运行的是该目录下某个名为 `ls` 的程序，而不是系统程序 `/bin/ls`。不要在 PATH 的开头或结尾添加冒号，也不要使用中间没有路径的一对冒号，因为这样会造成与点号(.)同样的效果。另外，最好将 `/usr/bin` 与 `/bin` 这样的系统目录置于 PATH 的开头，这样本地脚本就不会覆盖同名的系统默认程序。因此，应当使用的语法是

```
PATH=$PATH:${HOME}/bin
```

而不是

```
PATH=${HOME}/bin:$PATH
```

与工具相关的变量

很多系统工具都有自己的变量；`less` 将 `$LESS` 的值添加到 `less` 命令中。同样地，`ls` 将 `$LS_OPTIONS` 的值添加到 `ls` 命令中。通过设置这些环境变量，可以定义一些非常有用的简写命令。

```
# define tool-specific settings
Export LS_OPTIONS='--color=yes'
```

```
# Tidy up the appearance of less
Export LESS='-X'
```

less 还从\$LESS_TERMCAP_*变量中读取值, 这些变量含有关于终端功能的信息。下面几条语句很有用, 它们可以让手册页(用 less 来格式化)中的代码在运行时具有相应的颜色变化。



可从
wrox.com
下载源代码

```
# man pages in color
export LESS_TERMCAP_mb='${E[01;31m}'
export LESS_TERMCAP_md='${E[01;31m}'
export LESS_TERMCAP_me='${E[0m}'
export LESS_TERMCAP_se='${E[0m}'
export LESS_TERMCAP_so='${E[01;44;33m}'
export LESS_TERMCAP_ue='${E[0m}'
export LESS_TERMCAP_us='${E[01;32m}'
```

variables

还有一些广为人知的变量, 它们被很多其他工具用来让系统适应用户的需求。可以用这些变量为某些工具(例如 mail)指定使用哪个文本编辑器。还可以指定首选的分页工具——less 和 more 就是两个最常用的分页工具。

```
# define preferred tools
export EDITOR=vim
export PAGER=less
```

用户脚本可以使用这些变量来获得更强的灵活性。脚本中使用\${EDITOR:-vim}语句的意思是, 如果存在变量\$EDITOR 则使用原始值, 否则为应用程序提供一个默认值。



可从
wrox.com
下载源代码

```
#!/bin/bash
${EDITOR:-vim} "$1"
echo "Thank you for editing the file. Here it is:"
${PAGER:-less} "$1"
```

edit.sh

该脚本用首选的编辑器(\$EDITOR)编辑文件, 然后用首选的分页工具(\$PAGER)对其分页。

2.3.2 别名

别名为使用频率高但是不好记的命令提供了简写形式。在命令不使用配置文件或环境变量时, 别名还可以用于指定默认选项集。这些别名可以放到启动脚本中便于日常使用。

1. less

less 有一个-X 选项, 用于阻止程序结束后刷新屏幕。该选项很大程度上只是为了满足某些个人偏好。如果希望在 less 显示完文件后文件内容在终端中依然可见, 则需要使用-X 选项来阻止屏幕刷新(这与用 cat 显示文件很相似——命令结束后文件内容依然可见)。然

而,如果希望能查看运行 `less` 之前终端中显示的内容,则不要使用 `-X` 选项。试试这两种情况,看看更喜欢哪种。如果更倾向于使用 `-X`,则可以在 `~/.bashrc` 文件中设置一个别名。

```
alias less='less -X'
```

因为 `less` 命令从之前提到的 `$LESS` 环境变量中提取参数,所以也可以使用这个变量。

2. `cp`、`rm` 和 `mv` 别名

因为一些 Linux 发行版——特别是 RedHat,因而也包括 CentOS 和 Oracle Enterprise Linux——都是 RedHat Enterprise Linux 的二进制兼容的克隆版本,所以它们为 `cp`、`rm` 和 `mv` 命令精心定义了一些别名。这些别名都使用了 `-i` 选项,在交互式 shell 中删除或覆盖文件时询问是否确认操作。这是一个非常有用的安全特性,但很快这样不停的询问就变得让人生厌。如果不喜欢默认别名,可以在 `~/.bashrc` 中将它们删除。命令 `unalias rm` 就删除了 `rm` 别名。类似地, `unalias cp` 和 `unalias mv` 都让 `cp` 和 `mv` 恢复到原来的标准运行方式。



如果已知某个命令(如 `rm`)是别名,可以用两种方法来使用原始命令。如果知道命令的完整路径是 `/bin/rm`,可以输入 `/bin/rm` 来绕过别名机制。另一个简单的方法是在命令前面加上反斜线; `\rm` 将调用非别名的 `rm` 命令。

3. `ls` 别名

因为 `ls` 是一个非常常用的命令,所以有很多流行的 `ls` 别名。最常见的是表示 `ls -l` 的 `ll` 与表示 `ls -a` 的 `la`。您使用的发行版可能已经设置了这些别名。一些流行的 `ls` 别名包括下面这些:

```
# save fingers!
alias l='ls'
# long listing of ls
alias ll='ls -l'
# colors and file types
alias lf='ls -CF'
# sort by filename extension
alias lx='ls -lXB'
# sort by size
alias lk='ls -lSr'
# show hidden files
alias la='ls -A'
# sort by date
alias lt='ls -ltr'
```

4. 其他简写命令

还有很多其他频繁使用并希望使用别名的命令。在图形会话中,我们可以在打开 Web

浏览器的同时让其打开特定站点。

```
# launch webpages from terminal
alias bbc='firefox http://www.bbc.co.uk/ &'
alias sd='firefox http://slashdot.org/ &'
alias www='firefox'
```

另一个经常使用的命令是 `ssh`。该命令的使用有时简单如 `ssh hostname`，但有时它会用于相当复杂的命令行，此时别名就显得非常有用。



可从
wrox.com
下载源代码

```
# ssh to common destinations by just typing their name
# log in to 'declan'
alias declan='ssh declan'
# log in to work using a non-standard port (222)
alias work='ssh work.example.com -p 222'
# log in to work and tunnel the internal proxy to localhost:80
alias workweb='ssh work.example.com -p 222 -L 80:proxy.example.com:8080'
```

aliases

5. 修改命令行历史

`shell` 另一个可以在个性化设置中进行修改的特性是 `history` 命令。该特性受一些环境变量与 `shell` 选项(`shopt`)的影响。当同时有多个 `shell` 窗口打开，或者不同系统的同一用户登录到多个会话时，`history` 记录命令的方式可能变得有些复杂，并且一些历史命令可能会被新的命令覆盖。可以设置 `histappend` 选项来防止覆盖的发生。

`history` 另一个潜在的问题是如果个人文件没有较大的磁盘配额，它会占据大量的磁盘空间。`HISTSIZE` 变量定义 `shell` 会话在命令历史文件中最多能存储的命令条数；`HISTFILESIZE` 定义历史文件的最大尺寸。

`HISTIGNORE` 是一个用冒号隔开的命令列表，列表中的命令不会存储在历史文件中。它们通常是常用的 `ls` 等命令，对于审核来说一般无关紧要。不过从审核的角度而言，应当保持对 `rm`、`ssh` 与 `scp` 等这样的命令的监控。另外，`HISTCONTROL` 告诉 `history` 忽略前导空格，例如 `rm` 与 “ `rm` ” (命令前面有空格)都被存储为 `rm`，而不将它们区分开：

```
$ rm /etc/hosts
$ rm /etc/hosts
```

`HISTCONTROL` 还可以用于忽略重复命令，因此如果某个命令运行了多次，则可能没有必要在命令历史文件中记录多次运行的信息。`HISTCONTROL` 可以设置成 `ignorespace`、`ignoredups` 或者 `ignoreboth`。`~/.bashrc` 中与历史相关的一段设置可能如下所示：



可从
wrox.com
下载源代码

```
# append, don't overwrite the history
shopt -s histappend

# control the size of the history file
export HISTSIZE=100000
```

```
export HISTFILESIZE=409600

# ignore common commands
export HISTIGNORE=":pwd:id:uptime:resize:ls:clear:history:"

# ignore duplicate entries
export HISTCONTROL=ignoredups
```

history

6. ~/.inputrc 和/etc/inputrc

/etc/inputrc 和 ~/.inputrc 是 GNU 的 `readline` 工具(bash 与很多其他实用工具用它来从终端读取文本行)用来控制 `readline` 行为的文件。只有使用 `readline` 库的 shell(bash、dash 和 zsh)才使用这些配置文件,而其他 shell(ksh、tcsh 等)则不使用它们。它们还定义了很多有用的配置,这些配置正是 bash 优于 Bourne shell 的地方,如现今 PC 键盘上光标键的用法。一般没有必要编辑或创建自己的 ~/.inputrc(全局配置文件/etc/inputrc 通常就够用了)。了解文件的内容就能更好地理解 shell 与键盘命令的交互方式。inputrc 还定义了 8 比特的特性。如果使用 7 比特系统比较吃力,则可能需要使用这一特性。

应当知晓的另一个有用的 bash 选项是:

```
set completion-ignore-case on
```

它表示当输入 `cd foo` 并按下 Tab 键时,如果没有匹配 `foo*` 的目录,那么 shell 会不区分大小写地搜索目录,名称匹配 `Foo*`、`fOo*` 或 `fOO*` 的任何目录都会显示出来。

另一个 bash 选项可以关闭响铃:

```
set bell-style visible
```

应当注意的是, `inputrc` 会影响任何使用 `readline` 库的程序。这种处理方法一般是有好处的,因为这样可以使多个不同工具的行为保持一致性。我从来没有发现 `inputrc` 的这种处理方式会遇到什么问题,但如果要对 `inputrc` 进行一些修改,则会对其他使用 `readline` 库的程序造成一些影响,我们应当对这样的影响做到心中有数。

7. ~/.wgetrc 和/etc/wgetrc

如果需要使用代理服务器, ~/.wgetrc 文件可以用来为 `wget` 工具设置代理。例如:

```
http_proxy = http://proxyserver.intranet.example.com:8080/
https_proxy = http://proxyserver.intranet.example.com:8080/
proxy_user = steve
proxy_password = letmein
```

还可以在 shell 中定义相同的变量。

首先处理的是/etc/wgetrc 文件,但它会被用户的 ~/.wgetrc(如果存在)取代。



我们必须使用 `chmod 0600 ~/.wgetrc` 来处理要使用的 `~/.wgetrc` 文件——这是出于保护目的。有效的密码不应对用户以外的其他任何人可见！只要该文件的权限比 0600 稍稍宽松一些，`wget` 就会忽略它。

8. vi 模式

有 Unix 背景的人可能更习惯使用 `ksh`，无论是交互式使用还是用于 `shell` 脚本编程。对于交互式使用而言，`ksh` 通过 `Esc-k` 组合键来回滚到先前使用的命令，通过 `Esc-/` 组合键来搜索历史命令。这两个功能与 `bash` 的向上箭头键(或者 `^P`)和 `Ctrl-R` 组合键大致相同。如果要让 `bash`(或者 Unix 下的 Bourne shell)更像 `ksh`，则要设置 `-o vi` 选项：

```
bash$ set -o vi
bash$
```

2.3.3 vim 设置

下面这些有用的命令可以设置在 `~/.vimrc` 中，或者在 `vim` 的命令模式下手动设置。注意，`vim` 使用双引号(")来表示注释。这些例子表示的含义都是不言自明的。它们也可以在 `vim` 会话中交互式地进行设置，所以输入 `:syntax on` 或者 `:syntax off` 将为当前会话开启或关闭语法高亮功能。最好将所有常用的设置都预定义在 `~/.vimrc` 中。



可从
wrox.com
下载源代码

```
$ cat ~/.vimrc
" This must be first, because it changes other options as a side effect.
set nocompatible

" show line numbers
set number

" display "-- INSERT --" when entering insert mode
set showmode

" incremental search
set incsearch
" highlight matching search terms
set hlsearch
" set ic means case-insensitive search; noic means case-sensitive.
set noic
" allow backspacing over any character in insert mode
set backspace=indent,eol,start
" do not wrap lines
set nowrap

" set the mouse to work in the console
set mouse=a
" keep 50 lines of command line history
```



```
set history=50
" show the cursor position
set ruler
" do incremental searching
set incsearch
" save a backup file
set backup

" the visual bell flashes the background instead of an audible bell.
set visualbell

" set sensible defaults for different types of text files.
au FileType c set cindent tw=79
au FileType sh set ai et sw=4 sts=4 noexpandtab
au FileType vim set ai et sw=2 sts=2 noexpandtab

" indent new lines to match the current indentation
set autoindent
" don't replace tabs with spaces
set noexpandtab
" use tabs at the start of a line, spaces elsewhere
set smarttab

" show syntax highlighting
syntax on

" show whitespace at the end of a line
highlight WhitespaceEOL ctermbg=blue guibg=blue
match WhitespaceEOL /\s\+$/
```

vimrc

2.4 本章小结

操作系统、shell 与编辑器的选择有很多种。一般而言，编辑器的选择属于个人偏好。尽管对于 shell 脚本编程而言，现今很多开发环境(所有的 GNU/Linux 发行版、cygwin 与一些商业 Unix，如著名的 Solaris)都使用 GNU bash 与 bc、grep、ls 和 diff 等标准 Unix 工具的 GNU 实现，但是操作系统的选择在某种程度上说仍然意义重大。本书主要使用 GNU/Linux、bash 与 GNU 工具，但大多数内容都能适用于非 GNU 版本。

希望本章第二部分介绍的关于自定义的方法能帮助您将系统环境调整到符合您的个人喜好。应当让计算机适应人类，而不是反其道而行之，所以使用别名表示某个无须记住的复杂语法，这样可以让用户将注意力集中于真正想做的事情，而不是记住某些晦涩命令的确切语法。

前两章属于介绍性章节，其内容应当足够让我们准备好开始进行脚本编程。本书第 I 部分的余下章节将介绍一些可供使用的工具及其使用方法。本书后面的部分将介绍一些实用诀窍，我们可以使用这些诀窍编写脚本来解决现实生活中遇到的问题。

变 量

对于编程而言，没有变量将举步维艰：不能计数、循环或者从用户或环境中读取输入，也不能对任何东西进行修改。不用变量最多只能写出基本的批处理脚本(先执行一条命令，然后是下一条，依次下去)。有了变量，我们可以基于变量的状态来修改脚本的行为，也可以修改变量自身来反映脚本外部环境的变化。

本章将介绍 shell 中变量的使用方法，以及变量赋值与读取的语法，还将介绍一些最常见的标准预设 shell 变量和有用的 shell 选项。尽管其中一些变量只限于 `bash`，但大多数还是可通用于所有 shell。第 7 章深入介绍了一些 `bash`(和其他 shell)的高级功能，如数组和更强大的参数扩展特性。

3.1 使用变量

变量是可以通过引用名称来存储与检索任意数据的内存块。不必对所需的内存进行显式的分配，也不必在用完之后释放内存。尽管有些编程语言有着复杂的垃圾回收特性，但是 shell 脚本通常运行所需的数据量相对较小，而且运行周期较短，所以简单的垃圾回收模型就足够了。

shell 使用变量的语法有些独特。很多语言(如 Perl、PHP 等)只要在变量被引用时就使用 `$` 符号作为前缀。其他语言(如 Java 或 C)则不使用特定的标记来识别变量，从上下文就足以判断代码引用的是变量。



引用变量有时需要 `$` 符号(`echo $variable`)，有时又不能使用 `$` 符号(`variable=foo`)。有时需要用花括号将变量名括起来(`echo ${variable}bar`)，有时又不必这样做(`echo $variable bar`)。以上这些规则看似具有随意性，实则有其道理。不用担心，这些规则理解起来并不困难。

在 shell 中引用变量存储的值时，必须将\$符号置于变量名之前：

```
$ echo $PATH
```

当向变量写入值时，则不需要\$符号(下面的\$符号只是提示符，不是变量引用语法的一部分)：

```
$ PATH=/usr/sbin:/usr/bin:/sbin:/bin
```

也就是说，我们可以引用变量的名称，也可以在需要变量的值时通过\$符号引用变量的值，但是名称本身只会被当成普通字符串，除非是在赋值语句中的时候：

```
$ YOUR_NAME=steve
$ echo "The variable YOUR_NAME is $YOUR_NAME"
The variable YOUR_NAME is Steve
```

3.1.1 类型

在大多数语言中，变量都与某个“类型”联系起来，这个类型可能是字符串、整型、布尔型、浮点型或者其他类型。有些语言是强类型语言，它不允许整型与浮点型进行比较，或者将字符串向数字变量赋值等(可能让用户将它们转换为合适的类型，这样至少可以让程序员意图更加清晰)。shell 完全没有“类型”的概念。如果一定要说类型，那么任何变量都是字符串，但是有一些函数在处理字符串时会将其包含的数字字符当成数值来处理。在其他计算机语言中，这种错误会在编译期捕获。因为 shell 脚本是被解释而不是被编译，所以无法捕获到这种错误。

shell 变量的另一个特别之处是在使用它们之前不必显式地进行声明——没有赋值的变量就等于包含空字符串的变量。引用未定义的变量不会抛出错误。

```
$ cat unset.sh
#!/bin/bash

echo "The variable YOUR_NAME is $YOUR_NAME"
YOUR_NAME="Steve"
echo "The variable YOUR_NAME is $YOUR_NAME"
$ ./unset.sh
The variable YOUR_NAME is
The variable YOUR_NAME is Steve
$
```

3.1.2 变量的赋值

对变量赋值的方式主要有 3 种：

- 显式定义：VAR=value
- 读取：read VAR
- 命令替换：VAR=`date`、VAR=\$(date)

1. 显式定义: VAR=value

可以用语法 `x=y` 来定义变量的值(如果环境中不存在变量,这样做会创建变量),如之前的 `unset.sh` 示例中所示。要特别注意这条语法,因为经常会出现添加空格的错误,导致语法含义完全改变。

等于符号两边不允许有空格。这可能会使具有其他语言背景的人不适应,因为他们所使用的语言对于是否使用空格没有区别——空格一般用来使代码更清晰、更可读。`shell` 的这种语法是有原因的,下面几段将进行介绍。下面 3 个例子显示了等于(=)符号两边使用空格的 3 种不同方式,并说明它们是非法的变量赋值的原因。

```
variable = foo
```

上面的代码被当成是带有两个参数的命令(`variable`): 参数是 `=` 和 `foo`。其语法和 `ls -l foo` 完全一致。没有规定指出命令不可以使用 `=` 符号作为其第一个参数进行调用,所以 `shell` 无法确定语句是代表变量赋值。不过似乎并没有某个命令名为 `variable`,因而这成为了语句是赋值语句的理由。然而,我们并不知道脚本将要运行的系统上可能有什么命令,而且也不希望仅仅是因为某个最终用户创建了一个名为 `variable` 的文件而导致脚本的崩溃。

```
variable =foo
```

同样,上面的语法与 `ls =foo` 完全对等,所以它不是变量赋值。

```
variable= foo
```

该语法是一个较为常用的技术的特例。如果要在某个环境下运行一个命令,而不改变调用 `shell` 的环境,可以在命令之前加上变量赋值:

```
LD_LIBRARY_PATH=/usr/mozilla/lib firefox
```

像这样调用 Firefox,运行时 `LD_LIBRARY_PATH` 变量的值为 `/usr/mozilla/lib`(寻找系统库文件的默认路径是 `/usr/lib`)。所以, `LD_LIBRARY_PATH=firefox` 会使用空的 `LD_LIBRARY_PATH` 来调用 Firefox,这正是 `variable= foo` 语法。命令 `foo` 被调用时变量 `variable` 的值为空。

一旦用正确的语法对变量进行了赋值,就可以用 `$` 符号作为前缀来访问变量:

```
$ variable=foo
$ echo $variable
foo
$
```

2. 读取: read VAR

使用 `read` 命令可以交互式地对变量赋值:

```
$ cat first.sh
#!/bin/bash

read myvar
echo "myvar is $myvar"
```

如果运行该脚本，它会提示一行输入，并将变量 `myvar` 赋值为用户的输入。

可以通过首先回显一条提示让脚本更人性化。`echo` 的 `-n` 开关告诉 `echo` 在末尾不输出换行符，所以提示语和输入都在同一行显示：

```
echo -n "Enter your name: "
read myvar
echo "Hello $myvar"
```

交互式运行如下所示：

```
$ ./first.sh
Enter your name: Steve
Hello Steve
$ ./first.sh
Enter your name: Steve Parker
Hello Steve Parker
```

注意，输入的一整行都被读入到变量 `myvar` 中——也可以使用 `read` 一次性读入多个变量：

```
echo -n "Please enter your first name and last name: "
read firstname lastname
echo "Hello, $firstname. How is the $lastname family?"
```

上面的代码将读取两个变量，并且忽略任何空格。行中的最后一个变量将读取输入行中还没有读取的任何文本，所以这恰好能处理复姓的情况：

```
$ ./firstlast.sh
Please enter your first name and last name: Steve Parker Smith
Hello, Steve. How is the Parker Smith family?
$
```

然而，它对于输入过少的情况处理欠佳——变量 `lastname` 会一直存在于环境中(可以通过在脚本中添加 `set | grep name=` 来查看)，但值为空字符串：

```
$ cat firstlast.sh
#!/bin/bash

echo -n "Please enter your first name and last name: "
read firstname lastname
echo "Hello, $firstname. How is the $lastname family?"

echo "Relevant environment variables:"
set | grep "name="
$ ./firstlast.sh
Please enter your first name and last name: Steve Parker
Hello, Steve. How is the Parker family?
```

```

Relevant environment variables:
firstname=Steve
lastname=Parker
$ ./firstlast.sh
Please enter your first name and last name: Steve
Hello, Steve. How is the family?
Relevant environment variables:
firstname=Steve
lastname=
$

```

在上面的代码中, `the` 与 `family` 之间有两个空格, 所以脚本显示 `How is the $lastname family`, 但是这两个空格之间是一个长度为 0 的字符串 `$lastname`。本章后面将深入介绍这种情况。

3. 从文件读取

我们可以使用 `read` 命令从文件(实际上, 从终端读取也是从文件读取, 因为 Unix 下一切皆文件)读取文本行。下面的代码更清晰地展示了这一做法:

```

$ read message < /etc/motd
$ echo $message
Linux goldie 2.6.32-5-amd64 #1 SMP Fri Oct 15 00:56:30 UTC 2010 x86_64
$

```

然而, `/etc/motd` 文件的内容不止一行。下面的代码将文本行读入到变量 `message` 中, 一直循环直到不再有输入(如果读取到文件末尾, `read` 会返回非零值, 于是 `while` 循环就会停止——第 6 章将详细介绍循环)。

```

$ while read message
> do
>   echo $message
> done < /etc/motd
Linux goldie 2.6.32-5-amd64 #1 SMP Fri Oct 15 00:56:30 UTC 2010 x86_64

```

```

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the individual
files in /usr/share/doc/*/copyright.

```

```

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent permitted
by applicable law.
$

```

上面的命令将 `/etc/motd` 的内容导入到循环中, 所以循环不断地从 `/etc/motd` 中读取下一行(并将内容回显出来), 直到文件中的所有消息都被读取完。在循环的每一步中引入一个短暂的暂停可以更好地展示这种逐行读取的特性。

```
$ while read message
> do
>   echo $message
>   sleep 1
>   date
> done < /etc/motd
```

这次花了 8 秒来显示 8 行文本，但是通过书本很难展示时间的流逝，所以每次还运行了 `date` 命令来证明随着每行读取，时间确实在往前走：

```
Linux goldie 2.6.32-5-amd64 #1 SMP Fri Oct 15 00:56:30 UTC 2010 x86_64
Mon Oct 25 19:49:32 BST 2010

Mon Oct 25 19:49:33 BST 2010
The programs included with the Debian GNU/Linux system are free software;
Mon Oct 25 19:49:34 BST 2010
the exact distribution terms for each program are described in the
Mon Oct 25 19:49:35 BST 2010
individual files in /usr/share/doc/*/copyright.
Mon Oct 25 19:49:36 BST 2010

Mon Oct 25 19:49:37 BST 2010
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
Mon Oct 25 19:49:38 BST 2010
permitted by applicable law.
Mon Oct 25 19:49:39 BST 2010
```

4. 命令替换：VAR=`date`、VAR=\$(date)

变量赋值的另一种常见方式是将其值设置为某个给定命令的输出。这其实是第一种赋值方式 `VAR=value` 的变种。如果希望某个变量在星期一时值为 **Monday**，星期二时为 **Tuesday** 等，我们可以对 `date` 命令使用 `%A` 标志。该标志告诉 `date` 当前的区域设置下输出今天是星期几(第 14 章将详细讨论 `date` 命令)。



大多数 shell 还允许 `VAR=$(date)` 语法，但是原始的 Bourne shell 不允许。

```
$ cat today.sh
#!/bin/bash

TODAY=`date +%A`
echo "Today is $TODAY"
$ ./today.sh
Today is Monday
$
```

3.1.3 位置参数

当调用 shell 脚本时，对其传递参数有很大的作用——例如，参数是让脚本处理的某个

文件的名称。这些参数在脚本中是通过它们在命令行中所处的位置来引用的：\$0 是命令本身的名称，\$1 是第一个参数，\$2 是第二个参数，依此类推。这些变量的值无法修改，它们属于变量的特殊情况(一般而言，变量不能以数字开头，所以 shell 根本就不会将 1=hello 解释为变量赋值)。

典型的脚本用 `basename` 命令去掉位置参数的路径信息，所以 `./params.sh`、`/usr/local/bin/params.sh` 和 `~/bin/params.sh` 都被转换为 `params.sh`：

```
$ cat params.sh
#!/bin/bash

echo "My name is `basename $0` - I was called as $0"
echo "My first parameter is: $1"
echo "My second parameter is: $2"
$ ./params.sh one two
My name is params.sh - I was called as ./params.sh
My first parameter is: one
My second parameter is: two
```

脚本并不知道被调用时使用了多少个参数。上面的脚本假设有两个参数，但是对参数数目进行检测也很重要。这可以让脚本更人性化，因为它可以在用户不了解适当用法时给出用法信息。变量 \$# 告诉脚本调用时使用参数的数目，如下例所示(if 语句更常见的是写成 `if ["$#" -ne "2"]` 而不是 `if ["$#" -ne "2"]`，但在本例中很明显)。

```
$ cat params.sh
#!/bin/bash

echo "My name is `basename $0` - I was called as $0"
echo "I was called with $# parameters."
if [ "$#" -eq "2" ]; then
    # The script was called with exactly two parameters, good. Let's continue.
    echo "My first parameter is: $1"
    echo "My second parameter is: $2"
else
    # The "$#" variable must tell us that we have exactly two parameters.
    # If not, we will tell the user how to run the script.
    echo "Usage: `basename $0` first second"
    echo "You provided $# parameters, but 2 are required."
fi
$ ./params.sh one two
My name is params.sh - I was called as ./params.sh
I was called with 2 parameters.
My first parameter is: one
My second parameter is: two
$ ./params.sh one two three
My name is params.sh - I was called as ./params.sh
I was called with 3 parameters.
Usage: params.sh first second
You provided 3 parameters, but 2 are required.
```


以上做法不会出现什么问题，除非要对脚本进行扩展：

```
My eighth parameter is: $8
My ninth parameter is: $9
My tenth parameter is: $10
```

变量\$0到\$9被定义过，但是\$10不存在，且它被解释成\$1后面跟一个0(即使这与其他变量情况不同)。

```
$ cat params.sh
#!/bin/bash

echo "My name is `basename $0` - I was called as $0"
echo "My first parameter is: $1"
echo "My second parameter is: $2"
echo "....."
echo "My eighth parameter is: $8"
echo "My ninth parameter is: $9"
echo "My tenth parameter is: $10"
$ ./params.sh one two three four five six seven eight nine ten eleven twelve
My name is params.sh - I was called as ./params.sh
My first parameter is: one
My second parameter is: two
.....
My eighth parameter is: eight
My ninth parameter is: nine
My tenth parameter is: one0
$
```

必须有接收大于9个参数的方法：一般假设如果使用参数，\$0~\$9就够用了，但是如果处理更多参数，则显式地写成\$10、\$11、\$12、\$13等就显得很累赘——因为实际需要的是“下一个参数”。shift内置命令每次调用时将所有参数进行一次移位，首先截去\$1，然后是\$2，再然后是\$3等。无法将移位逆向进行，所以应当确保在调用shift之前已完全处理过将被移除的参数。

```
$ cat manyparams.sh
#!/bin/bash

echo "My name is `basename $0` - I was called as $0"
echo "I was called with $# parameters."
count=1
while [ "$#" -ge "1" ]; do
    echo "Parameter number $count is: $1"
    let count=$count+1
    shift
done
$ ./manyparams.sh one two three
My name is manyparams.sh - I was called as ./manyparams.sh
```

```

I was called with 3 parameters.
Parameter number 1 is: one
Parameter number 2 is: two
Parameter number 3 is: three
$ ./manyparams.sh one two three four five six seven eight nine ten eleven twelve
My name is manyparams.sh - I was called as ./manyparams.sh
I was called with 12 parameters.
Parameter number 1 is: one
Parameter number 2 is: two
Parameter number 3 is: three
Parameter number 4 is: four
Parameter number 5 is: five
Parameter number 6 is: six
Parameter number 7 is: seven
Parameter number 8 is: eight
Parameter number 9 is: nine
Parameter number 10 is: ten
Parameter number 11 is: eleven
Parameter number 12 is: twelve
$

```

可以使用 `shift n` 一次截去多个变量。所以如果要截去 3 个变量, `shift 3` 等于 `shift; shift; shift`。不过后者更为常见, 因为那样更具可移植性, 且表达式的意图更清晰。

所有的参数

最后两个读取传递的参数的变量是 `$*` 和 `$@`。它们非常相似, 而且经常容易混淆。如下面的代码所示, 无论输入什么, 前面 4 行看起来都几乎一样, 除了在用双引号传递给脚本时引号中的空格被保留, 并且在被脚本处理时它们也被双引号引用。所以, `five` 前的多个空格总是被忽略, 因为它们在传递给 `shell` 时没有用双引号引用。只有当脚本在双引号(最后一例)中处理 `$@` 时, `two` 和 `three` 之间的多个空格才被保留。

```

$ cat star.sh
#!/bin/bash

echo Dollar Star is $*
echo "Dollar Star in double quotes is $*"
echo Dollar At is @$
echo "Dollar At in double quotes is @"
echo
echo "Looping through Dollar Star"
for i in $*
do
    echo "Parameter is $i"
done
echo
echo "Looping through Dollar Star with double quotes"
for i in "$*"
do

```

```

    echo "Parameter is $i"
done
echo
echo "Looping through Dollar At"
for i in $@
do
    echo "Parameter is $i"
done
echo
echo "Looping through Dollar At with double quotes"
for i in "$@"
do
    echo "Parameter is $i"
done
$ ./star.sh one "two three" four five
Dollar Star is one two three four five ← 空格未保留。
Dollar Star in double quotes is one two three four five ← 引用的空格被保留。
Dollar At is one two three four five ← 空格未保留。
Dollar At in double quotes is one two three four five ← 引用的空格被保留。

Looping through Dollar Star
Parameter is one ← 不引用$, 每个单词被当成独立的单词。
Parameter is two
Parameter is three
Parameter is four
Parameter is five

Looping through Dollar Star with double quotes
Parameter is one two three four five ← 使用"$*", 整个参数列表被当成一个参数。

Looping through Dollar At
Parameter is one ← 不引用$, $@与$*一样。
Parameter is two
Parameter is three
Parameter is four
Parameter is five

Looping through Dollar At with double quotes
Parameter is one ← "$@"保留调用者的假设。"two three"是一个参数, 且保留空格。
Parameter is two three
Parameter is four
Parameter is five
$

```

第4章将介绍参数本身包含特殊字符的情况。

3.1.4 返回码

在 Unix 与 Linux 中, 每个命令都返回一个 0~255 之间的代码——也就是 1 个字节(尽管-1 回绕变成了 255、-2 回绕变成了 254 等)。返回的代码可以暗示程序运行成功与否, 有时也可以提供更详细的信息。shell 将变量\$?设置为上一次运行命令返回的代码。例如, grep

运行失败的方式有很多——最常见的是文件中没有匹配的字符串：

```
$ grep nutty /etc/hosts
$ echo $?
1
$
```

或者发生错误，如文件不存在或不可读等。

```
$ grep goldie /etc/hosttable
grep: /etc/hosttable: No such file or directory
$ echo $?
2
$
```

对于 `grep`，返回 1 表示“不匹配”，大于或等于 2 表示 `grep` 本身运行的某种错误，如文件没有找到。

如果命令运行成功——运行无错误且有匹配的文本行——那么按照惯例，返回退出码 0，表示运行成功：

```
$ grep goldie /etc/hosts
192.168.1.13 goldie
$ echo $?
0
$
```

这在根据其他命令的运行结果来采取不同措施的情况下很有用。有时可能需要在尝试某种操作失败后采取正确的措施。其他情况下，命令运行失败并没有关系，因为只是想知道它能否成功运行。

1. 后台进程

虽然后台进程要到第10章才详细介绍，但此处值得一提的是，我们可以在后台运行进程，并使用变量 `$!` 来找到其进程 ID(PID)。这对于跟踪后台进程很有用。一定要注意，如果不小心可能会出现竞争：

```
#!/bin/sh
ls -R /tmp &
sleep 10
strace -p $!
```

在这个(假设的)例子中，如果 `ls` 的运行时间超过 10 秒，则会发现使用 `strace` 对 `ls` 的运行跟踪了 10 秒。然而，如果 `ls` 运行少于 10 秒，那它的 PID 就会过期——当前的进程树中可能没有这样一个 PID，所以 `strace` 会失败。还有可能发生的是，特别是在高负荷系统上，当进程处于休眠状态时，另一个进程可能会被分配相同的 PID，最后导致跟踪的是完全不相关的进程。

2. 同时读取多个变量

本章前面简要介绍过，我们可以在一条语句中读取多个变量。下面这个脚本从数据文件中读取第一行，然后将单词赋值给变量：

```
$ cat datafile
the quick brown fox
$ read field1 field2 field3 < datafile
$ echo Field one is $field1
Field one is the
$ echo Field two is $field2
Field two is quick
$ echo Field three is $field3
Field three is brown fox
```

如果文件中的输入不够，则有些变量将为空：

```
$ echo the quick > datafile
$ read field1 field2 field3 < datafile
$ echo Field one is $field1
Field one is the
$ echo Field two is $field2
Field two is quick
$ echo Field three is $field3
Field three is
$
```

技巧

一个常用技巧是使用下面的语法来设置\$one=1、\$two=2、\$three=3和\$four=4：

```
echo 1 2 3 4 | read one two three four
```

它的运行方式与期望的不同：只要管道本身与调用 shell 没有完成变量赋值，那么用管道连接的 read 命令就一直运行。

使用 while 读取

下面的脚本从/etc/hosts 的每一行读取前两个单词。由于 readline 的工作方式所致，变量 aliases 读取所有的别名。注意，该脚本对于非正常输入的处理欠佳，例如注释行(如下面的 IP is # - its name is The)。grep -v "^#" | while read 则能更好地处理这种情况。

```
$ while read ip name alias
> do
>   if [ ! -z "$name" ]; then
>     # Use echo -en here to suppress ending the line;
>     # aliases may still be added
>     echo -en "IP is $ip - its name is $name"
>     if [ ! -z "$aliases" ]; then
>       echo " Aliases: $aliases"
```

```

>     else
>         # Just echo a blank line
>     echo
>     fi
> fi
> done < /etc/hosts
IP is 127.0.0.1 - its name is localhost Aliases: spo
IP is # - its name is The Aliases: following lines are desirable for IPv6
capable hosts
IP is ::1 - its name is localhost Aliases: ip6-localhost ip6-loopback
IP is fe00::0 - its name is ip6-localnet
IP is ff00::0 - its name is ip6-mcastprefix
IP is ff02::1 - its name is ip6-allnodes
IP is ff02::2 - its name is ip6-allrouters
IP is 192.168.1.3 - its name is sky
IP is 192.168.1.5 - its name is plug
IP is 192.168.1.10 - its name is declan Aliases: declan.steve-parker.org
IP is 192.168.1.11 - its name is atomic
IP is 192.168.1.12 - its name is jackie
IP is 192.168.1.13 - its name is goldie Aliases: smf sgp
IP is 192.168.1.227 - its name is elvis
IP is 192.168.0.210 - its name is dgoldie Aliases: intranet ksgp
$

```

3.1.5 删除变量

有时需要删除变量。这会释放变量占据的内存，当然也会影响到接下来的变量引用。删除的命令是 `unset`，使用方法如下：

```

$ echo $myvar

$ myvar=hello
$ echo $myvar
hello
$ unset myvar
$ echo $myvar

$

```

将变量赋值为空字符串也能实现变量的删除，不过后面将会介绍，这与使用 `unset` 不完全相同：

```

$ myvar=hello
$ echo $myvar
hello
$ myvar=
$ echo $myvar

$

```



有些变量无法删除；它们被称为“只读”变量。我们无法删除\$1、\$2 和 \$#等变量。

还有一些变量(RANDOM、SECONDS、LINENO、HISTCMD、FUNCNAME、GROUPS 和 DIRSTACK)可以删除，但是删除后就无法恢复它们之前的特殊功能(\$RANDOM 删除后再也不能返回随机数等)。

注意，第一行代码在赋值之前就访问了变量 `myvar`，但是当执行 `echo $myvar` 命令时并没有报错。`shell` 中没有变量声明的概念：因为变量没有类型，也就没有必要指定 `int`、`char` 或 `float` 等，所以变量在第一次赋值时是被隐式地声明。访问不存在的变量只会返回空字符串或 0，具体取决于上下文。

`shell` 脚本的这一特点让它比大多数语言要简单明了，但这要付出很大的代价：例如在调试脚本时，查找拼写错误的变量名非常困难。下面的代码计算直角三角形的斜边长度，它的错误在哪里？

```
$ cat hypotenuse.sh
#!/bin/sh

# calculate the length of the hypotenuse of a Pythagorean triangle
# using hypotenuse^2 = adjacent^2 + opposite^2
echo -n "Enter the Adjacent length: "
read adjacent
echo -n "Enter the Opposite length: "
read opposite
osquared=$(( $opposite ** 2 ))          # get o^2
asquared=$(( $adjacent ** 2 ))          # get a^2
hsquared=$(( $osquared + $asquared ))    # h^2 = a^2 + o^2
hypotenuse=`echo "scale=3;sqrt ($hsquared)" | bc`
# bc does sqrt
echo "The Hypotenuse is $hypotenuse"
```

这类错误很难预防。在本例中，我们知道是拼写错误。如果我们不知道是拼写错误或者语法错误，那么可能会很难跟踪。更糟糕的是，有时甚至注意不到有错误发生，因为脚本运行并产生输出时并不会报错。

代码中的错误在下面的这一行中(先不谈这个简单的脚本只考虑了整数情况)：

```
hsquared=$(( $osquared + $asquared ))
```

`$osquared` 应该是 `$osquared`。没有 `$osquared` 这个变量，所以它默认被 0 代替。结果是对边长度完全起不到作用，代码实现的只是将邻边的长度平方，然后加上 0，最后将它们之和开方，所以总是返回邻边的原始值。

3.2 预定义变量和标准变量

`shell` 本身还提供了一些变量。这些变量各有特点：有些是纯信息类型的，如 `$BASH_`

VERSION 表示运行的 bash 的版本(例如, 4.1.5(1)-release), 但可以通过赋值来覆盖它们。其他变量根据当前环境返回不同的值, 如\$PIPESTATUS 返回前一次管道中命令的返回值, 其他的则根据各自的功能返回不同值(如\$SECONDS 的返回值每过 1 秒增加 1)。另外, 如果对某些变量进行赋值(或删除), 它们会失去其特殊含义: \$RANDOM 会一直返回随机数, 直到给它赋值, 之后将不再产生随机数。\$SECONDS 将总是不停地计数; 如果对它赋值 SECONDS=35, 然后在 5 秒后读取, 将返回 40。还有一些变量完全不可写, 例如如果试图给 UID 赋值, 则会得到错误信息。最后一些变量, 如 TIMEFORMAT, 它们本身没有什么作用, 但被用来影响其他命令的工作方式。

下面介绍一些更为常见且有用的预定义变量, 以及一些不太常见的预定义变量。

3.2.1 BASH_ENV

BASH_ENV 是一个文件名称(可能是相对路径, 这时应当特别注意脚本运行的位置), 该文件在执行前要进行分析, 就如同在打开交互式 shell 前要分析 ~/.bashrc 一样。

3.2.2 BASHOPTS

BASHOPTS 是在 bash 4.1 中出现的新变量。它是所有启用的 shell 选项(shopts)的列表。使用 shopt|grep -w on 也能得到同样的信息, 但 BASHOPTS 更适合于计算机分析。它是只读变量, 可以通过内置命令 shopt 来改变它的输出(-s 用来开启选项, -u 用来关闭选项):

```
$ shopt mailwarn
mailwarn                off
$ echo $BASHOPTS
checkwinsize:cmdhist:expand_aliases:extquote:force_ignore:hostcomplete
:interactive_comments:progcomp:promptvars:sourcepath
$ shopt -s mailwarn
$ shopt mailwarn
mailwarn                on
$ echo $BASHOPTS
checkwinsize:cmdhist:expand_aliases:extquote:force_ignore:hostcomplete
:interactive_comments:mailwarn:progcomp:promptvars:sourcepath
$ shopt -u mailwarn
$ shopt mailwarn
mailwarn                off
$ echo $BASHOPTS
checkwinsize:cmdhist:expand_aliases:extquote:force_ignore:hostcomplete
:interactive_comments:progcomp:promptvars:sourcepath
$
```

bash 4.1 中有 40 个 shell 选项, 在 bash 手册页中都有详细的文档; 下面介绍一些有用且有趣的选项:

- checkhash——它检查某个文件在运行之前文件路径是否被缓存到哈希表中。
- checkwinsize——它在每个命令(准确地说是作为 PROMPT_COMMAND 的一部分)运行之后更新 LINES 和 COLUMNS 变量。这意味着如果调整窗口的大小, 然后运

行使用这些变量(如 `top`)的命令, 运行中的 `shell` 会自动获取新的窗口大小。如果没有这个选项, 则必须手动运行 `resize` 命令来获取新的窗口大小。

- `cmdhist`——它用来压缩多行命令, 例如:

```
$ for i in `seq 10 -1 1`
> do
>   echo -en "${i} ..."
> done ; echo "boom"
10 ...9 ...8 ...7 ...6 ...5 ...4 ...3 ...2 ...1 ...boom
$
```

上面的代码在 `shell` 的命令历史中会被压缩成一行:

```
for i in `seq 10 -1 1`; do echo -en "${i} ..."; done; echo "boom"
```

- `hostcomplete`——它是 `bash` 的一个有用技巧, 可以将命令补全扩展到主机名。如果我们要从 `atomic` 登录主机 `declan`, 可以输入:

```
steve@atomic:~$ ssh steve@de <tab>
```

然后 `bash` 会在 `/etc/hosts` 中找到 `declan`, 并将命令扩展为 `ssh steve@declan`。它作用于 `/etc/hosts`, 而不是 `DNS`; `DNS` 中的查找方式不可能一样。

- `login_shell`——它是只读变量; 如果当前 `shell` 是登录 `shell`, 那它就会被赋值。

3.2.3 SHELLOPTS

`SHELLOPTS` 与 `BASHOPTS` 类似; 它是 `-o` 选项集的列表。如果设置了 `-o vi`, 则 `vi` 会出现在选项列表中, 且 `shell` 将工作在 `vi` 模式下(第 2 章的 2.3 节有介绍)。与 `BASHOPTS` 一样, `SHELLOPTS` 也是只读变量。我们可以使用两种不同的方法对选项进行设置。有些选项只能用一种方法生效。在下面的代码中, 我们可以使用两种语法开启 `errexit` 特性(`-e` 或 `-o errexit`), 然后再关闭(`+e` 或者 `+o errexit`):

```
$ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
$ set -e
$ echo $SHELLOPTS
braceexpand:emacs:errexit:hashall:histexpand:history:interactive-comments:monitor
$ set +o errexit
$ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
$
```

同样, 这些选项在 `bash` 手册页的内置命令 `set` 部分有详细的文档, 但下面介绍一些更常用的选项。它们中的很多都可以在其他 `shell` 中使用:

- **-e / -o errexit**——如果命令返回非零退出状态码，则 shell 将退出。它适用的情况是，如果需要脚本中每条命令都执行成功，且认为一旦出错则退出是最安全的处理措施。
- **-f / -o noglob**——禁用路径名扩展。
- **-m / -o monitor**——如果该选项被设置(默认情况)，则当后台命令结束后，bash 在下次显示新的提示符时会给出一条信息：

```
$ ls /tmp &
[1] 2922
keyring-UDudcH
orbit-steve
OSL_PIPE_1000_SingleOfficeIPC_54f1d8557767a73f9bc36a8c3028b0
pulse-Mm0m5cufbNQY
ssh-EwfFww1963
svdlb.tmp
[1]+ Done                ls /tmp
$
```

- **pipefail**——它可以代替变量 PIPESTATUS。如果该选项关闭(默认情况)，则管道的返回码是管道最右边命令的非零退出状态。所以，如果管道在中途出错(没有以 192.167 开头的 IP 地址，则 grep 会失败，但是 cat 和 cut 命令能正常运行)，则很难判断 grep 是否运行成功：

```
$ cat /etc/hosts | grep 192.167 | cut -f1
$ echo $?
0
```

cut 命令运行成功，所以返回码是 0(表示成功)，但这可能不是真正需要的。然而，如果设置了 pipefail 选项，则可以从管道中的任何命令检测到错误：

```
$ set -o pipefail
$ cat /etc/hosts | grep 192.167 | cut -f1
$ echo $?
1
$
```

- **-o vi**——该选项将终端从 emacs 模式转到 vi 模式。
- **-x**——该选项会在每条命令运行前显示命令。这在调试 shell 脚本时特别有用：将 `#!/bin/sh -x` 置于脚本开头，或者在脚本中开启选项 `set -x(set +x 关闭选项)`，或者运行 `sh -x myscript.sh`。另外请参考本节后面的 BASH_XTRACEFD 变量。下面每一行命令前面都加上变量 PS4 的值(默认为+)：

```
$ cat x.sh
#!/bin/bash
echo "Hello, world!"
```

```
if [ "$?" -eq "0" ]; then
    # comments are ignored
    echo "Hurray, it worked!"
else
    echo "Oh no, echo failed!"
fi
$ sh -x x.sh
+ echo Hello, world!
Hello, world!
+ [ 0 -eq 0 ]
+ echo Hurray, it worked!
Hurray, it worked!
$
```

我们发现测试语句["\$?" -eq "0"]使用值进行了扩展，所以测试语句为[0-eq 0]。

3.2.4 BASH_COMMAND

BASH_COMMAND 是当前运行命令的名称。它在大多数时候用处不大，但是不包括 shell 内置命令(如 trap)在内。可以在 trap 调用中，用它显示中断时正在进行的操作。下面这个脚本运行了一些命令，并在按下^C 中断时显示脚本正在进行的操作。我们很可能在脚本休眠的时候捕获到脚本：

```
$ cat trap.sh
#!/bin/bash

trap cleanup 1 2 3 15

cleanup()
{
    echo "I was running \"$BASH_COMMAND\" when you interrupted me."
    echo "Quitting."
    exit 1
}

while :
do
    echo -en "hello. "
    sleep 1
    echo -en "my "
    sleep 1
    echo -en "name "
    sleep 1
    echo -en "is "
    sleep 1
    echo "$0"
done
$ ./trap.sh
hello. my name is ./trap.sh
```

```
hello. my ^CI was running "sleep 1" when you interrupted me.
Quitting.
$
```

如果删除 `sleep` 命令，则会捕获到 `echo` 命令，或者是 `while` 循环的测试条件：

```
$ ./trap.sh
hello. my name is ./trap.sh
hello. my name is ./trap.sh
hello. my name is ./trap.sh
^Chello. I was running "echo -en "hello. "" when you interrupted me.
Quitting.
$
```

3.2.5 BASH_SOURCE、FUNCNAME、LINENO 和 BASH_LINENO

`BASH_SOURCE`、`FUNCNAME`、`LINENO` 和 `BASH_LINENO` 是相当有用的调试变量，它们可以告诉我们脚本中的当前位置信息，甚至是在使用多个文件的情况下。`LINENO` 只给出在脚本中当前所处的行号：

```
$ cat lineno.sh
#!/bin/bash

echo "Hello, World"
echo "This is line $LINENO"
$ ./lineno.sh
Hello, World
This is line 4
$
```

这对调试很有用，特别是在从最终用户获取有用信息的时候。脚本可以给出具体的行号，而不是报错 `Error occurred in the fourth debugging point`，这样可以对于脚本问题的诊断进行精确定位。然而，得知当前所处函数名称也很有用，且 `FUNCNAME` 能给出更详细的信息。它还能给出整个函数调用栈，显示如何从一开始通过函数调用至此：

```
$ cat funcname.sh
#!/bin/bash

function func1()
{
    echo "func1: FUNCNAME0 is ${FUNCNAME[0]}"
    echo "func1: FUNCNAME1 is ${FUNCNAME[1]}"
    echo "func1: FUNCNAME2 is ${FUNCNAME[2]}"
    echo "func1: LINENO is ${LINENO}"
    func2
}

```

```
function func2()
{
    echo "func2: FUNCNAME0 is ${FUNCNAME[0]}"
    echo "func2: FUNCNAME1 is ${FUNCNAME[1]}"
    echo "func2: FUNCNAME2 is ${FUNCNAME[2]}"
    echo "func2: LINENO is ${LINENO}"
}

func1

$ ./funcname.sh
func1: FUNCNAME0 is func1
func1: FUNCNAME1 is main
func1: FUNCNAME2 is
func1: LINENO is 8
func2: FUNCNAME0 is func2
func2: FUNCNAME1 is func1
func2: FUNCNAME2 is main
func2: LINENO is 17
$
```

所以在 `func2` 中, 我们看到 `FUNCNAME[0]` 是当前所处函数的名称, `FUNCNAME[1]` 是上层调用函数, `FUNCNAME[2]` 是调用 `func1` 的函数(实际上是主脚本, 所以名称比较特殊, 为 `main`)。这些都比较有用且很有趣, 但遇到包含不同函数的多个库文件的情况会如何?

```
$ cat main1.sh
#!/bin/bash
. lib1.sh
. lib2.sh

func1

$ cat lib1.sh
function func1()
{
    echo "func1: FUNCNAME0 is ${FUNCNAME[0]}"
    echo "func1: FUNCNAME1 is ${FUNCNAME[1]}"
    echo "func1: FUNCNAME2 is ${FUNCNAME[2]}"
    echo "func1: BASH_SOURCE0 is ${BASH_SOURCE[0]}"
    echo "func1: BASH_SOURCE1 is ${BASH_SOURCE[1]}"
    echo "func1: BASH_SOURCE2 is ${BASH_SOURCE[2]}"
    echo "func1: LINENO is ${LINENO}"
    func2
}

$ cat lib2.sh
function func2()
{
    echo "func2: FUNCNAME0 is ${FUNCNAME[0]}"
    echo "func2: FUNCNAME1 is ${FUNCNAME[1]}"
```

```

echo "func2: FUNCNAME2 is ${FUNCNAME[2]}"
echo "func2: BASH_SOURCE0 is ${BASH_SOURCE[0]}"
echo "func2: BASH_SOURCE1 is ${BASH_SOURCE[1]}"
echo "func2: BASH_SOURCE2 is ${BASH_SOURCE[2]}"
# This comment makes lib2.sh different from lib1.sh
echo "func2: LINENO is ${LINENO}"
}

$ ./main1.sh
func1: FUNCNAME0 is func1
func1: FUNCNAME1 is main
func1: FUNCNAME2 is
func1: BASH_SOURCE0 is lib1.sh
func1: BASH_SOURCE1 is ./main1.sh
func1: BASH_SOURCE2 is
func1: LINENO is 9
func2: FUNCNAME0 is func2
func2: FUNCNAME1 is func1
func2: FUNCNAME2 is main
func2: BASH_SOURCE0 is lib2.sh
func2: BASH_SOURCE1 is lib1.sh
func2: BASH_SOURCE2 is ./main1.sh
func2: LINENO is 10
$

```

本例中，func1 中 FUNCNAME[0] 是 func1，BASH_SOURCE[0] 是 lib1.sh，LINENO 行是 lib1.sh 文件的第 9 行。类似地，func2 中 BASH_SOURCE[0] 是 lib2.sh，FUNCNAME[0] 是 func2，且 LINENO 是 lib2.sh 文件的第 10 行。单独使用 LINENO 而不使用 BASH_SOURCE 完全没有意义，因为这样必须将文件名称硬编码到 echo 语句本身中去，并且有可能还需要对行号进行硬编码——函数可以很容易找到其在不同文件中不同时间所处的位置，并且可以确定在某些位置上，我们不会去注意在错误消息中显示的是一些不同的文件。

```

$ cat err1.sh
#!/bin/bash
. elib1.sh
. elib2.sh

func1
$ cat elib1.sh
. errlib.sh
function func1()
{
    err $LINENO this is func1, does it get it right?
    func2
}
$ cat elib2.sh
. errlib.sh
function func2()

```

```

{
    err $LINENO this is func2, does it get it right?
}
$ cat errlib.sh
function err()
{
    echo
    echo "*****"
    echo
    echo -en "error: Line $1 in function ${FUNCNAME[1]}"
    echo "which is in the file ${BASH_SOURCE[1]}"
    shift
    echo "error: Message was: $@"
    echo
    echo "*****"
    echo
}
$ ./err1.sh
*****
error: Line 4 in function func1 which is in the file elib1.sh
error: Message was: this is func1, does it get it right?
*****

*****
error: Line 4 in function func2 which is in the file elib2.sh
error: Message was: this is func2, does it get it right?
*****
$

```

本例中的 `errlib.sh` 提供了一个有用且通用化的错误报告库，可以被很多不同的脚本作为通用调试工具使用。

`LINENO` 必须传递给 `err()` 函数，因为它总是在变化。与另外两个变量不同，它不是数组(如果 `LINENO` 是数组，则能像引用 `FUNCNAME[1]` 和 `BASH_SOURCE[1]` 一样引用 `LINENO[1]`)。当向 `err()` 函数传递参数时，可能还会添加自定义的错误消息。自定义的消息还能够帮助用户理解之前的输出。然而，还可以进行最终的优化：`bash` 添加了 `BASH_LINENO` 变量，它是数组；`bash` 手册页上说明 “`${BASH_LINENO[$i]}` 是源文件中调用 `${FUNCNAME[$i]}` 所在的行号(或者如果在另一个 `shell` 函数中引用，则是 `${BASH_LINENO[$i - 1]}`)”，所以 `${BASH_LINENO[0]}` 给出相关的行号：

```

$ cat errlib2.sh
function err()
{
    echo
    echo "*****"
    echo
    echo -en "error: Line ${BASH_LINENO[0]} in function ${FUNCNAME[1]} "
    echo "which is in the file ${BASH_SOURCE[1]}"
    echo "error: Message was: $@"
}

```

```

echo
echo "*****"
echo
}
$

```



如果删除这3个变量中的任意一个，它们都会失去其特殊作用。

3.2.6 SHELL

SHELL 不总是被 `bash shell` 定义——尽管可能预期是这样！如果当 `bash` 启动时该变量已经被定义，那么它不会被修改，所以不要想当然地认为它会被修改：

```

$ SHELL=/bin/strangeshell bash
$ echo $SHELL
/bin/strangeshell
$

```

要检测是否在 `bash` 下运行，下面是更好的测试代码，尽管变量 `BASH_VERSION` 可能已经在任何 `shell` 中被定义过：

```

if [ -z "$BASH_VERSION" ]; then
    echo "This is not really Bash"
else
    echo "Yes, we are running under Bash - version $BASH_VERSION"
fi

```

3.2.7 HOSTNAME 和 HOSTTYPE

`HOSTNAME` 是主机名称。`HOSTTYPE` 是主机类型。``uname -n``和``uname -m``的结果可能更加可靠，且不能被用户修改。

3.2.8 工作目录

`PWD` 是当前工作目录，可以用``pwd``获取。`OLDPWD`是上一个工作目录。这可以省去改变工作目录之前记录``pwd``的工作。命令`cd -`可以将工作目录退回到`$OLDPWD`。

3.2.9 PIPESTATUS

`PIPESTATUS` 存储上次运行的管道或命令的退出状态的数组。注意，下面使用 `PIPESTATUS` 的方法不会起作用：

```

echo $a | grep $b | grep $c
echo "The echo of $a returned ${PIPESTATUS[0]}"
echo "The grep for $b returned ${PIPESTATUS[1]}"
echo "The grep for $c returned ${PIPESTATUS[2]}"

```


PIPESTATUS[0]起了作用，但是对于接下来的调用，PIPESTATUS 被成功运行的 echo 命令重新赋值。必须在管道结束后立即获取整个数组，否则接下来的 echo 语句被当成一个单命令的管道，且会覆盖之前的 PIPESTATUS 数组。查看是否有命令运行错误的技巧如下：

```
ls $dir | grep $goodthing | grep -v $badthing
echo ${PIPESTATUS[*]} | grep -v 0 > /dev/null 2>&1
if [ "$?" -eq "0" ]; then
    echo "Something in the pipeline failed."
else
    echo "Only good things were found in $dir, no bad things were found. Phew!"
fi
```

3.2.10 TIMEFORMAT

time 命令的输出一般都是采用机器可读的形式，但描述不会太详细。可以另外编写脚本来解释与格式化 time 的输出数据，但对于大多数可能遇到的场景，变量 TIMEFORMAT 就可以做到。它的格式与 printf 类似，百分号(%)表示由适当值代替的特殊字符。所以，%U 表示进程在用户模式(非操作系统调用)下的运行时间，%S 表示系统模式下的运行时间，而 %R 表示一共消耗的时间。它们 3 个都可以通过添加 l(字母 L 的小写)来将时间从秒扩展到分和秒，或者使用精度修改符(0~3)来指定秒精确到几位小数。

time 的标准格式如下：

```
$ time ls > /dev/null
real    0m0.007s
user    0m0.001s
sys     0m0.001s
```

实际上使用的是下面的格式，因为如果没有定义 TIMEFORMAT，则 bash 使用下面的格式。

```
'\nreal\t%31R\nuser\t%31U\nsys\t%31S'
```

如果 TIMEFORMAT 为空，time 根本不显示输出。

POSIX 的 time 格式类似，且通过 -p 开关来使用：

```
$ time -p ls > /dev/null
real 0.00
user 0.00
sys 0.00
```

如果要用更可读的格式来显示 CPU 的使用情况，TIMEFORMAT 变量可以按如下所示设置。

```
$ TIMEFORMAT="%21U user + %21S system / %21R elapsed = %P% CPU Utilisation"
$ time sleep 2
0m0.00s user + 0m0.00s system / 0m2.00s elapsed = 0.04% CPU Utilisation
```

```
$ time ls -R /var > /dev/null
0m0.07s user + 0m0.05s system / 0m1.30s elapsed = 9.79% CPU Utilisation
$ time ls >/dev/null
0m0.00s user + 0m0.00s system / 0m0.00s elapsed = 100.00% CPU Utilisation
$
```

类似地，TIMEFORMAT 可以用比标准 time 输出更自然的方式显示总共的运行时间：

```
$ TIMEFORMAT="%U user + %S system = %lR total elapsed time"
$ time ls -R /var > /dev/null 2>&1
0.036 user + 0.020 system = 0m0.056s total elapsed time
$
```

sleep 不会消耗用户或系统时间，所以时间不会增加。剩下的时间必须用来等待系统对任务进行处理：

```
$ time sleep 1
0.001 user + 0.000 system total = 0m1.002s total elapsed time
$ time expr 123 \* 321
39483
0.001 user + 0.002 system = 0m0.043s total elapsed time
```

从/dev/urandom 读取 512MB 数据会消耗大量的系统时间。这是因为/dev/urandom 提供的随机数据是从环境中产生的，且要产生足够随机化的数据可能比较费时间。

```
$ time dd if=/dev/urandom of=/tmp/randomfile bs=1024k count=512
512+0 records in
512+0 records out
536870912 bytes (537 MB) copied, 63.5919 seconds, 8.4 MB/s
0.003 user + 56.113 system = 1m3.614s total elapsed time
$
```

3.2.11 PPID

PPID 是调用 shell 或 shell 脚本的进程的 ID。\$\$ 是另一个特殊变量，它提供 shell 自身的进程 ID。\$\$ 经常用于创建临时文件，因为对于创建随机的、唯一的文件名，这样基本上是安全的。当运行脚本的多份副本时，\$\$ 还能被脚本用来识别自身。



mktemp(在第 12 章中介绍)通常是更好的创建临时文件的方式。

```
$ cat pid.sh
#!/bin/bash
echo "Process $$: Starting up with arguments $@ for my parent, $PPID"
sleep 10
$ ./pid.sh 1 & ./pid.sh 2 & ./pid.sh 3
```

```
[1] 2529
[2] 2530
Process 2531: Starting up with arguments 3 for my parent, 2484
Process 2529: Starting up with arguments 1 for my parent, 2484
Process 2530: Starting up with arguments 2 for my parent, 2484
[1]- Done                      ./pid.sh 1
$
[2]+ Done                      ./pid.sh 2
$
```

3.2.12 RANDOM

RANDOM 产生 0~32767 之间的随机数。下面的脚本产生 10 个 200~500 之间的随机数。

```
$ cat random.sh
#!/bin/bash
MIN=200
MAX=500
let "scope = $MAX - $MIN"
if [ "$scope" -le "0" ]; then
    echo "Error - MAX is less than MIN!"
fi

for i in `seq 1 10`
do
    let result="$RANDOM % $scope + $MIN"
    echo "A random number between $MIN and $MAX is $result"
done
$ ./random.sh
A random number between 200 and 500 is 462
A random number between 200 and 500 is 400
A random number between 200 and 500 is 350
A random number between 200 and 500 is 279
A random number between 200 and 500 is 339
A random number between 200 and 500 is 401
A random number between 200 and 500 is 465
A random number between 200 and 500 is 320
A random number between 200 and 500 is 290
A random number between 200 and 500 is 277
$
```

脚本首先计算随机数的跨度，然后使用模运算符使值处于跨度之中。之后加上\$MIN 的值生成所需范围中的数。

3.2.13 REPLAY

REPLAY 是在没有为 read 命令提供变量情况下的默认变量。

```
$ read
hello world
$ echo $REPLY
hello world
```

3.2.14 SECONDS

SECONDS 返回 shell 运行的(整)秒数。在 shell 脚本中，它是脚本运行的时间，而不是调用它的 shell 的运行时间。如果将 SECONDS 修改为其他整数，它会从那个整数开始计时。将 SECONDS 赋值为一个非整数值会导致其为 0。如果删除 SECONDS，它会从此失去其特殊功能且成为一个普通变量，即使稍后重新对它进行设置。

SECONDS 除了计时之外还有其他作用：如果 shell 脚本临时需要一个唯一且不是完全可预测的数，则总是可以用(sleep 1; echo \$SECONDS)来获取一个当前脚本之前没有使用过的数。SECONDS 的另一个用法是缘自这样一个事实，命令 timeout(1)返回它执行的命令的退出码，如果超时则返回 124。所以根本无法得知命令是否真的超时，或者本身就是返回 124。SECONDS 可以协助确定命令是否超时：

```
#!/bin/bash

SECONDS=0
timeout 60s slow_command
timeout_res=$?
# 124 if timedout, but 124 could be the return code from slow_command
if [ "$SECONDS" -lt "60" ]; then
    # it did not time out; the value is from slow_command.
    echo "The command did not time out; it returned after $SECONDS seconds."
    cmd_res=$timeout_res
else
    # It timed out; take special action here
    echo "The command timed out."
fi
```

3.2.15 BASH_XTRACEFD

BASH_XTRACEFD 是 bash 4.1 中新引入的变量。当使用 -x 特性时，该变量允许脚本自己定义输出到什么文件(默认输出到 stderr)。下面的脚本还包含了一个获取新文件描述符的技巧，这在无法跟踪已经打开多少文件的情况下很有用。我们在脚本中使用 set -x 和 set +x 来开启脚本中这两个语句之间 shell 的 -x 特性。

```
$ cat xtrace.sh
#!/bin/bash

TRACER=/tmp/tracer.txt
TRACEFD=3
# Find the next available file descriptor
```

```
ls -l /proc/$$/fd
while [ -e /proc/$$/fd/$TRACEFD ] && [ $TRACEFD -lt 255 ]; do
    let "TRACEFD += 1"
done

if [ $TRACEFD -eq 254 ]; then
    echo "Error: No more file descriptors available!"
    exit 1
fi

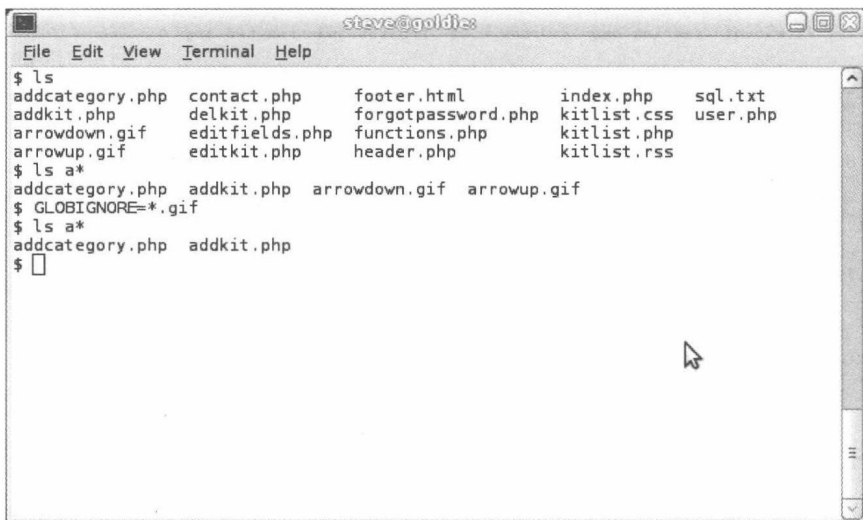
echo "FD is $TRACEFD"

eval "exec $TRACEFD>$TRACER"
BASH_XTRACEFD=$TRACEFD
ls -l /proc/$$/fd
# Enable logging with -x
set -x
date
echo hello world
sleep 1
date
set +x
# disable logging
eval "exec $TRACEFD>&-"
echo "The result of our tracing was in $TRACER:"
cat $TRACER
$ ./xtrace.sh
total 0
lrwx----- 1 steve steve 64 Nov 26 16:53 0 -> /dev/pts/4
lrwx----- 1 steve steve 64 Nov 26 16:53 1 -> /dev/pts/4
lrwx----- 1 steve steve 64 Nov 26 16:53 2 -> /dev/pts/4
lr-x----- 1 steve steve 64 Nov 26 16:53 255 -> /home/steve/book/part1/variables/xt
race.sh
FD is 3
total 0
lrwx----- 1 steve steve 64 Nov 26 16:53 0 -> /dev/pts/4
lrwx----- 1 steve steve 64 Nov 26 16:53 1 -> /dev/pts/4
lrwx----- 1 steve steve 64 Nov 26 16:53 2 -> /dev/pts/4
lr-x----- 1 steve steve 64 Nov 26 16:53 255 -> /home/steve/book/part1/variables/xt
race.sh
l-wx----- 1 steve steve 64 Nov 26 16:53 3 -> /tmp/tracer.txt
Fri Nov 26 16:53:27 GMT 2010
hello world
Fri Nov 26 16:53:28 GMT 2010
The result of our tracing was in /tmp/tracer.txt:
+ date
+ echo hello world
+ sleep 1
+ date
+ set +x
$
```

本例中，必须使用 `eval` 与 `exec` 以便让脚本以正确的方式分析变量的值。如果已经得知要写入的文件的描述符，我们可以直接使用 `exec 5>$TRACER`。

3.2.16 GLOBIGNORE

GLOBIGNORE 是用于 `bash` 的通配特性的特殊变量，用来忽略通配符的某些模式。第4章将详细介绍通配符。与很多列表格式的 `shell` 变量(如 `PATH`、`GLOBOPTS` 等)一样，它是用冒号隔开的正则表达式列表。如果某个文件能匹配 GLOBIGNORE 中的模式，则该文件被当成不存在。如图 3-1 所示，目录中包含一些 HTML、PHP、TXT、RSS 和 CSS 文件。如果要列出以 `a` 开头的文件，则将只看到前 4 个文件。然而当我们通过设置 GLOBIGNORE 来忽略匹配模式 `*.gif` 的文件，则相同的 `ls` 命令只会显示两个 PHP 文件，不会显示 GIF 文件。



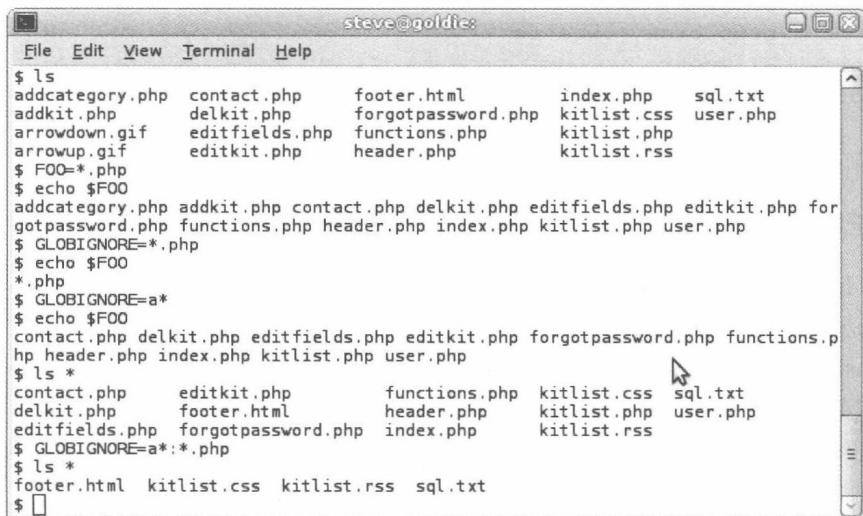
```

steve@goldie:
File Edit View Terminal Help
$ ls
addcategory.php  contact.php  footer.html  index.php  sql.txt
addkit.php      delkit.php  forgotpassword.php  kitlist.css  user.php
arrowdown.gif   editfields.php  functions.php  kitlist.php
arrowup.gif     editkit.php   header.php    kitlist.rss
$ ls a*
addcategory.php  addkit.php  arrowdown.gif  arrowup.gif
$ GLOBIGNORE=*.gif
$ ls a*
addcategory.php  addkit.php
$ 

```

图3-1

图 3-2 所示的例子更清晰地显示了 GLOBIGNORE 的作用：



```

steve@goldie:
File Edit View Terminal Help
$ ls
addcategory.php  contact.php  footer.html  index.php  sql.txt
addkit.php      delkit.php  forgotpassword.php  kitlist.css  user.php
arrowdown.gif   editfields.php  functions.php  kitlist.php
arrowup.gif     editkit.php   header.php    kitlist.rss
$ FOO=*.php
$ echo $FOO
addcategory.php addkit.php contact.php delkit.php editfields.php editkit.php forgotpassword.php functions.php header.php index.php kitlist.php user.php
$ GLOBIGNORE=*.php
$ echo $FOO
*.php
$ GLOBIGNORE=a*
$ echo $FOO
contact.php delkit.php editfields.php editkit.php forgotpassword.php functions.php header.php index.php kitlist.php user.php
$ ls *
contact.php  editkit.php  functions.php  kitlist.css  sql.txt
delkit.php  footer.html  header.php    kitlist.php  user.php
editfields.php  forgotpassword.php  index.php  kitlist.rss
$ GLOBIGNORE=a*:*.php
$ ls *
footer.html  kitlist.css  kitlist.rss  sql.txt
$ 

```

图3-2

在图 3-2 中，第一个 `ls` 命令显示目录中的所有文件。将 `FOO` 赋值为 `*.php` 意味着 `echo $FOO` 将匹配所有的 `.php` 文件。`GLOBIGNORE=*.php` 让 `echo $FOO` 的行为看起来像是找不到匹配的文件——输出其未扩展的原始值，即字符串 `*.php`。

设置 `GLOBIGNORE=a*`, 则 `echo $FOO` 将忽略所有以 `a` 开头的文件。设置 `GLOBIGNORE=a*:*.*.php`, 则忽略匹配任一模式的文件。

该变量适用于源控制系统。如图 3-3 所示，一些文件用~扩展名进行备份。

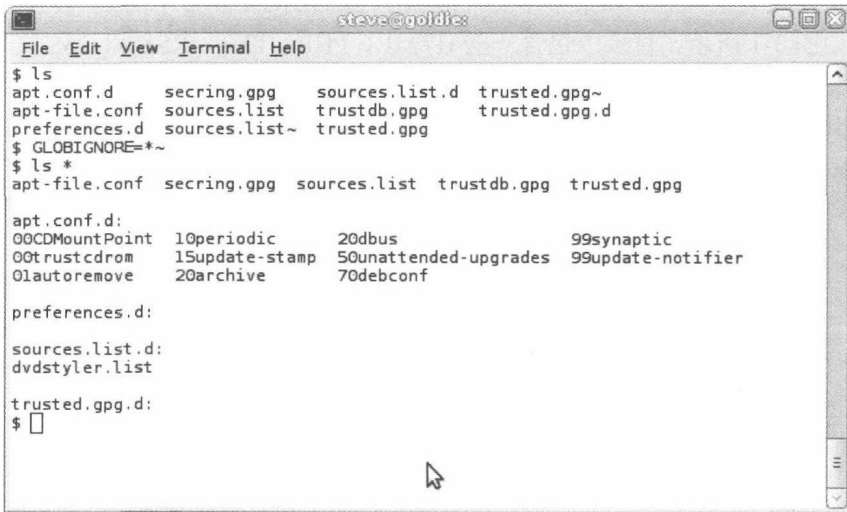


图3-3

3.2.17 HOME

HOME 是当前用户的主目录的路径。如果单独运行 `cd`，则会将目录改变到 `$HOME`。在 `bash shell` 中，`$HOME` 也能用 `~` 表示，所以下面的命令都使用了变量 `HOME` 的值：

```
cd ; grep proxy .wgetrc
cd $HOME ; grep proxy .wgetrc
grep proxy ~/.wgetrc
grep proxy ${HOME}/.wgetrc
```

且都会对当前用户的`~/.wgetrc` 文件运行 `grep`。

3.2.18 IFS

IFS 是 Internal Field Separator 的缩写：它列出当成空格使用的字符的集合。它的默认值是 `<space><tab><newline>`。回过头来看看本章之前的命令 `read VAR`，我们发现 `read firstname lastname` 能用来读取第一个单词(`firstname`)，之后可以有也可以没有单词(余下部分都被读取到变量 `lastname`)。这是由 IFS 造成的，如果将 IFS 修改成其他值，则会改变 `read` 命令的运行结果。这很有用，例如在读取被冒号隔开的 `/etc/passwd` 文件或者读取用点号隔开的 IP 地址时：

```

$ cat ifsl.sh
#!/bin/bash

# Save the original IFS
oIFS=$IFS
IFS=":"
# /etc/passwd is delimited by colons only.
while read login pass uid gid name home shell
do
    # Ignore those with /bin/false, or home directories in /var
    if [ "$shell" != "/bin/false" ] && [ ! -z "${home%\var/*}" ]; then
        echo "User $login ($name) lives in $home and uses `basename $shell`"
    fi
done < /etc/passwd

# Not necessary as we're exiting the script, but it is good practice;
# subsequent commands will want the normal IFS values.
oIFS=$IFS
$ ./ifsl.sh
User root (root) lives in /root and uses bash
User daemon (daemon) lives in /usr/sbin and uses sh
User bin (bin) lives in /bin and uses sh
User sys (sys) lives in /dev and uses sh
User sync (sync) lives in /bin and uses sync
User games (games) lives in /usr/games and uses sh
User proxy (proxy) lives in /bin and uses sh
User nobody (nobody) lives in /nonexistent and uses sh
User steve (Steve Parker,,,) lives in /home/steve and uses bash
$

```

在显示输出时，\$IFS 中的第一个字符用来进行字符填充(另外，输入中的所有\$IFS 字符被删除)。这说明了如果要保留空格，则需要在 echo 语句中使用双引号。

```

$ echo hello          world
hello world
$

```

3.2.19 PATH

PATH 是用来查找程序文件的用冒号隔开的目录列表。查找顺序从左至右，所以如果 PATH=/home/steve/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11，则 ls(一般在/bin 中)被当成 /home/steve/bin/ls、/usr/local/bin/ls、/usr/local/bin/ls、/usr/bin/ls 查找，直到找到/bin/ls。bash shell 在哈希表中存储了这些值，所以下次调用 ls 时，shell 会记得是/bin/ls。如果希望使用自己创建的/home/steve/bin/ls，可以运行 hash -r 来强制 shell 清空哈希表。hash -l 可以列出哈希表中的路径。

PATH 中，点号(.)用来表示调用程序的当前目录。不太为人所知的双冒号(::)和点号作用一样，且处于 PATH 变量开头或末尾的单个冒号都会被扩展为当前工作目录。所以，如

果 `PATH=./usr/bin:/bin`，则调用 `ls` 时，当前目录中任何名为 `ls` 的文件都会被执行。这样导致的结果是，轻则运行结果不恰当(如果程序与预期的行为不一致)，重则成为重大安全问题(特别是以 `root` 身份登录)：攻击者将含有恶意代码的 `ls` 置于 `/tmp` 中，然后就等待超级用户显示 `/tmp` 中的内容。超级用户将在不知情的情况下运行攻击者的 `ls` 程序。这种尽量让自己看起来无害(完成的任务看似与 `root` 希望的一样)的程序就是所谓的特洛伊木马，以留在特洛伊城外著名的木马命名。

3.2.20 TMOUT

`TMOUT` 的用法有 3 种：用于内置命令 `read`、`select`，或者交互式 `bash shell`。如果它被删除或者等于 0，则将其忽略。如果 `TMOUT` 的数值为正，则使用它的前面 3 个命令在 `$TMOUT` 秒之后超时。

`read` 命令如果超时，则以一个大于 128 的退出码退出(这种情况下总是以返回码 142 退出)。

```
steve@atomic:~$ TMOUT=10
steve@atomic:~$ read -p Name: name
Name:steve@atomic:~ ← 提示符(steve@atomic)与“Name:”提示符显示在同一行。
$ echo $?           这是因为 read 命令超时，且 shell 接下来就显示提示符。
142
steve@atomic:~$ unset TMOUT
steve@atomic:~$ read -p Name: name
Name:Steve
steve@atomic:~$ echo $?
0
steve@atomic:~$ echo $name
Steve
steve@atomic:~$
```

我们还可以通过调用 `read -t timeout` 达到同样的目的：

```
steve@atomic:~$ unset TMOUT
steve@atomic:~$ read -t 5 -p Name: name
Name:steve@atomic:~$ echo $? ← 两个提示符又在同一行显示。
142
steve@atomic:~$
```

内置命令 `select` 也会在 `$TMOUT` 秒之后超时，但是并不会打断循环。在下面的代码中，分别选择 1、2 和 3 这些选项，如果超时则重新显示选项与 `#?` 提示符，且循环再次运行。然而，请注意 `select` 循环体(`echo foo is $foo`)没有运行。

```
steve@atomic:~$ TMOUT=5
steve@atomic:~$ select foo in one two three
> do
>   echo foo is $foo;
> done
```

```

1) one
2) two
3) three
#? 1
foo is one
#? 2
foo is two
#? 3
foo is three
#? 1) one
2) two
3) three
#? ^C
steve@atomic:~$

```

shell 会在 TMOUT 秒之后退出。这是个很有用的安全特性，特别是可以防止用户离开以 root 身份登录的终端。这种情况时有发生，但是很多系统管理员却不愿意公开承认。然而，要是在自己工作到一半的时候突然从会话中退出会令人非常讨厌(从会话中退出不一定总有明显的提示，特别是提示符没有包含主机名的时候)：

```

steve@atomic:~$ TMOUT=5
steve@atomic:~$ timed out waiting for input: auto-logout
Connection to atomic closed.
steve@declan:~$

```

如果系统设置的 TMOUT 比我们把它修改过来消耗的时间要短，那么我们的输入速度就必须相当快——TMOUT 监视的不是静止状态，而是命令的执行。如果 TMOUT 被赋值为小于 10，且输入 unset TMOUT 或 TMOUT=0 要用 10 秒，那情况就不妙了！



可以开启另一个不受 TMOUT 限制的 shell，或者使用 scp 之类的命令用好的配置文件覆盖不好的配置文件。

3.2.21 TMPDIR

TMPDIR 可用于任何临时文件。如果它没有被赋值，则使用 /tmp。除了 shell 本身，有些应用程序也使用 TMPDIR。所以，如果某个应用程序不断地向 /tmp 写入大量的文件，则应当为程序修改 TMPDIR，让别的目录作为其专用存储空间(可能是操作系统以外的外部存储空间)。

3.2.22 用户标识变量

能够识别运行脚本的用户通常也很有用。shell 启动后，UID 被自动赋值为用户的 ID 并且只读，所以用户无法修改。GROUPS 是存储当前用户的组 ID 的数组。GROUPS 数组可写，但是一旦它被赋值，则会失去其特殊性质。

注意，`$USER` 不是 `bash shell` 定义的变量。它可能被设置为正确的值，但也能设置为任意值：它对于 `bash shell` 没有特殊意义。不建议使用 `$USER` 来精确识别用户。

通常使用下面的方法在运行特权级脚本前检查用户是否为 `root` 用户。

```
#!/bin/bash
if [ "$UID" -ne "0" ]; then
    echo "Sorry, you are not root."
    exit 1
else
    echo "You are root - you may proceed."
fi
```

3.3 本章小结

与条件执行一样，变量是一门真正有用的语言所必需的特性之一。能够从环境中读取数据并以此作为运行不同代码的依据意味着，`shell` 已经成为真正有用的编程语言，而不只是一个批处理器。`shell` 与大多数语言有些不同，在于其赋值与变量读取的语法不同，还有 `shell` 没有变量类型的概念。

有些变量由系统预定义，其中一些(如 `UID`)只能读取，不能写入。其他变量(如 `RANDOM`)则是如果向它们赋值，将会失去其特殊含义。

第 7 章将深入讨论变量，其中包括了大量 `bash` 相关的变量。第 9 章介绍数组，它是一种特殊变量，可以在 `bash`、`ksh` 和 `zsh` 中使用。

通配符扩展

编写 shell 脚本时，通配符的用法有两种。shell 本身将通配符用于文件名扩展，所以我们可以用 `a*` 匹配所有以字母 `a` 开头的文件，用 `*.txt` 匹配所有文本文件。另外还有更加强大的正则表达式，它被很多如 `sed`、`awk` 与 `grep` 这样的 shell 实用程序使用。与语法简单的 shell 相比，这些实用程序使用更加正式的语言和语法。`bash` shell 提供的通配符扩展要比标准的 Bourne shell 更加强大，本章将对其进行介绍。

还有一些规则会影响跟 shell 有特殊关系的引用和转义字符，而这些规则与正则表达式也有关系。当需要将 `a*` 传递给 `sed` 时，可能不希望 shell 将 `a*` 扩展到匹配当前目录中的文件，而是想将通配符本身直接传递给 `sed`。本章的 4.2.2 节将介绍能实现这一目的各种技术与特殊字符。

4.1 文件名扩展(globbing)

globbing 这个不起眼的单词源自 Unix 的原作者之一 Dennis Ritchie 编写的命令 `/etc/glob`。`glob` 看似是 `global` 的缩写，因为它原本的设计是要搜索整个 `$PATH` 中的路径。但 `glob` 的原始实现只搜索 `/bin` 目录，这被认为是错误的。如今，`which` 命令代替了 `glob` 的作用，但是从 `glob` 这个命令，我们有了 `globbing` 这个单词，意思是“使用通配符扩展搜索文件”；而且它不引用 `PATH` 变量。

文件名扩展中的两个关键字符是问号(?)与星号(*)。问号匹配任意单个字符，星号匹配任意字符序列。因为，如果给定包含各种模式的文件集合，我们可以使用这些通配符找到匹配的文件。

```
$ ls
abc abcdef abcdefghijk abc.php abc.txt ABC.txt def mydoc.odt xyz.xml
ABC ABCDEF abcdef.odt abctxt abc.TXT alphabet DEF xyz
$ ls a*
abc      abcdefghijk abc.php  abc.txt  alphabet
abcdef  abcdef.odt  abctxt  abc.TXT
```

```
$ ls A*
ABC ABCDEF ABC.txt
$ ls A??
ABC
$ ls a??
abc
```

这一特性经常用来查找带有给定扩展名的所有文件。注意，`*txt` 不同于 `*.txt`。



尽管 Unix 与 Linux 广泛使用 `.txt`、`.sh` 或 `.conf` 作为文件名后缀，但扩展名本身不具有其他一些操作系统(特别是 Microsoft Windows)赋予它的特殊含义。我们可以将 OpenOffice 文档 `myconfiguration.odt` 重命名为 `myconfiguration.bin`，而且它还能用 OpenOffice 编辑。实用工具 `file` 根据文件内容来识别文件，而不会被文件扩展名误导。

```
$ ls *.txt
abc.txt ABC.txt
$ ls *.*??
abcdef.odt abc.php abc.txt abc.TXT ABC.txt mydoc.odt xyz.xml
$ ls *txt
abctxt abc.txt ABC.txt
$
```

因为 `*` 匹配任意字符序列，所以也可以匹配空字符。而问号则不同，它必须总是匹配一个字符。

```
$ ls a*b*
abc      abcdefghijk abc.php abc.txt alphabet
abcdef   abcdef.odt abctxt  abc.TXT
$ ls a?b*
ls: cannot access a?b*: No such file or directory
$
```

本例还说明了通配符可以处于任意位置，而不仅仅是文件名的两端。下面的命令列出包含字母 `h` 的所有文件。注意，`abc.php` 包括在内。前面已经提到，文件名中包含点号没有特殊意义。

```
$ ls *h*
abcdefghijk abc.php alphabet
$
```

尽管下面要介绍的不是文件名扩展，但 `bash shell` 另一个有用的特性是可以扩展一系列包含在花括号中的字符串。通过指定需要的数，文件列表 `/etc/rc*.d` 被更精确地匹配。

```
$ ls -ld /etc/rc{0,1,2,3,4,5,6}.d
drwxr-xr-x 2 root root 4096 Nov 25 19:38 /etc/rc0.d
drwxr-xr-x 2 root root 4096 Nov 25 19:38 /etc/rc1.d
drwxr-xr-x 2 root root 4096 Nov 25 19:38 /etc/rc2.d
drwxr-xr-x 2 root root 4096 Nov 25 19:38 /etc/rc3.d
drwxr-xr-x 2 root root 4096 Nov 25 19:38 /etc/rc4.d
drwxr-xr-x 2 root root 4096 Nov 25 19:38 /etc/rc5.d
drwxr-xr-x 2 root root 4096 Nov 25 19:38 /etc/rc6.d
$
```

这在创建具有类似结构的多个目录时特别有用：

```
$ mkdir -p /opt/weblogic/domains/domain{1,2,3}/bin
$ ls -ld /opt/weblogic/domains/*/bin
drwxrwxr-x 2 steve steve 4096 Sep 22 11:40 /opt/weblogic/domains/domain1/bin
drwxrwxr-x 2 steve steve 4096 Sep 22 11:40 /opt/weblogic/domains/domain2/bin
drwxrwxr-x 2 steve steve 4096 Sep 22 11:40 /opt/weblogic/domains/domain3/bin
$
```

尽管文件名中的点号没有特殊意义，但连字符(-)几乎被每一个 Unix 与 Linux 命令使用。当遇到一个文件名为-rf 或者是更糟糕的-rf.时，我们该如何删除它？答案似乎是 rm -rf.，但这会删除当前目录下的所有文件和目录。很多(尽管不是全部)命令使用双连字符(--)来表示命令选项的结束与要操作的文件名列表的开始。所以 rm -- "-rf."将删除文件-rf。用*或?命名的文件可能造成类似的问题——只要可能，最好避免在文件名中使用这些字符。

尽管?和*是文件名扩展的两个主要元字符，但还有[...]这样的结构。它有 3 种区别很小的用法。最常用的可能是可以用它来定义范围，所以 a?c.txt 能匹配 alc.txt 以及 abc.txt。我们可通过范围[a-z]来指定中间的字符必须是 a~z 之间的字母，如下所示：

```
$ ls a[a-z]c.txt
abc.txt
$
```

范围可以任意选择，但[a-z]与[0-9]使用得最多。另外有一些定义好的类能代替这两个范围，并且意义相同。[a-z]等同于[:alpha:]，[0-9]等同于[:digit:]。全部的可用类如表 4-1 所示。

表 4-1 可用类

类	成 员
alnum	A-Z、a-z、0-9
alpha	A-Z、a-z
blank	空格和制表符
cntrl	ASCII 字符 0-31(非打印控制字符)
digit	0-9
graph	A-Z、a-z、0-9 与标点符号

(续表)

类	成 员
lower	a-z
print	ASCII 字符 32-127(可打印字符)
punct	标点符号(A-Z、a-z 和 0-9 以外的可打印字符)
space	空格、制表符、LF(10)、VT(11)、FF(12)、CR(13)
upper	A-Z
xdigit	0-9、A-F、a-f

如果范围不够精确，我们还可以在括号中放置一系列字符来进行匹配。所以，[aeiou]将匹配任何元音字母，[13579]将匹配奇数数字等。我们可以在范围开头使用感叹号(!)或脱字符(^)来表示相反的意思，即不包含范围中的字符。

```
$ # files starting with a lowercase vowel
$ ls [aeiou]*
abc      abcdefghijk  abc.php  abc.txt  alphabet
abcdef  abcdef.odt   abctxt   abc.TXT
$ # files not starting with a lowercase vowel
$ ls ![aeiou]*
ABC  ABCDEF  ABC.txt  def  DEF  mydoc.odt  xyz  xyz.xml
$
```

最后，如果需要匹配字符[，我们必须将其置于列表的开头，如[[aeiou]。如果要匹配字符-，可以将它置于列表开头或结尾，但不能是其他位置，所以[-aeiou]或[-aeiou]都可以。

4.1.1 bash 的文件名扩展特性

有时，我们可能需要完全关闭文件名扩展功能。bash 指令 set -o noglob(或者 set -f)可以将其关闭，set +o noglob(或 set +f)重新开启。假设有两个.odt 文档，如果要创建名为*d*.odt 的文件，文件名扩展将把命令 touch *d*.odt 解释为 touch abcdef.odt mydoc.odt。注意，文件的时间戳被修改了。

```
$ ls -l *d*.odt
-rw-rw-r-- 1 steve steve 505 Nov 13 10:13 abcdef.odt
-rw-rw-r-- 1 steve steve 355 Dec 20 09:17 mydoc.odt
$ touch *d*.odt
$ ls -l *d*.odt
-rw-rw-r-- 1 steve steve 505 Dec 22 11:51 abcdef.odt
-rw-rw-r-- 1 steve steve 355 Dec 22 11:51 mydoc.odt
$
```

通过关闭文件名扩展可以得到想要的结果。下面的*d*.odt 则按照字面进行解释。

```
$ set -o noglob
$ touch *d*.odt
$ set +o noglob
$ ls -l *d*.odt
-rw-rw-r-- 1 steve steve 505 Dec 22 11:51 abcdef.odt
-rw-rw-r-- 1 steve steve 0 Dec 22 11:52 *d*.odt
-rw-rw-r-- 1 steve steve 355 Dec 22 11:51 mydoc.odt
$
```

4.1.2 shell 选项

shell 选项改变 shell 的工作方式。shell 的有些选项会影响到文件名扩展。分别使用命令 `shopt -s optionname` 与 `shopt -u optionname` 来设置与取消选项。如果要使用某个 shell 选项，可以使用 `shopt optionname` 查询是否设置，或者按照编程的方法使用 `shopt -q optionname`。该命令不显示任何输出，但如果选项被设置则返回 0，否则返回 1。

```
$ shopt nullglob
nullglob          off
$ shopt -q nullglob
$ echo $?
1
$ shopt -s nullglob
$ shopt nullglob
nullglob          on
$ shopt -q nullglob
$ echo $?
0
$
```

shell 用来防止意外修改的一个特性是所谓的“隐藏”文件——就是以点号开头的文件，如 `~/.bashrc`、`~/.ssh/` 等——它们不会被标准的文件名扩展匹配。当我们理解它的存在之后，会发现这确实是个很有用的特性，否则会让人很沮丧。为何 `rm -rf /tmp/myfiles/*` 不会将目录清空？因为有 `/tmp/myfiles/.config`。我们不能使用 `rm -rf /tmp/myfiles/.*`，因为这会匹配 `/tmp/myfiles/..`，表示 `/tmp` 目录本身。我们总能使用 `rm -rf /tmp/myfiles`，这会删除目录 `/tmp/myfiles` 以及目录中的全部内容；但如果希望保留目录，则可以使用 shell 选项 `dotglob`。

```
$ ls *
abc abcdef abcdefghijk abc.php abc.txt ABC.txt def *d*.odt xyz
ABC ABCDEF abcdef.odt abctxt abc.TXT alphabet DEF mydoc.odt xyz.xml
$ ls .*
.abc .abcdef .def

.:
abc abcdef abcdefghijk abc.php abc.txt ABC.txt def *d*.odt xyz
ABC ABCDEF abcdef.odt abctxt abc.TXT alphabet DEF mydoc.odt xyz.xml

...:
```



```

3204.txt eg glob.txt wildcard.txt wildcards.odt
$ shopt -s dotglob
$ ls *
abc  abcdef  abcdefghijk  abctxt  ABC.txt  .def  mydoc.odt
.abc .abcdef  abcdef.odt  abc.txt  alphabet  DEF  xyz
ABC  ABCDEF  abc.php  abc.TXT  def  *d*.odt  xyz.xml
$

```

shell 的另一个特性在某些情况下可能造成不便，那就是当模式不匹配任何文件时，它保持不变。所以对于上面列出的文件，**a***扩展为 **abc abcdefghijk abc.php abc.txt alphabet abcdef abcdef.odt abctxt abc.TXT**，但是 **b***扩展为包含通配符的字面字符串 **b***。我们可以使用 **nullglob** 选项强制将它扩展为空字符，这样就不会对通配符本身进行处理。

```

$ for filename in a* b*
> do
>   md5sum $filename
> done
674ea002ddbaf89619e280f7ed15560d abc
1d48a9f8e8a42b0977ec8746cd484723 abcdef
b7f0f386f706aelbc3c8fa3bffb0371c abcdefghijk
8116e5ba834943c9047b6d3045f45c8c abcdef.odt
ac100d51fbab3ca3677d59e63212cb32 abc.php
d41d8cd98f00b204e9800998ecf8427e abctxt
e45f6583e2a3feacf82d55b5c8ae0a60 abc.txt
a60b09767be1fb8d88cbb1afbb90fb9e abc.TXT
3df05469f6e76c3c5d084b41352fc80b alphabet
md5sum: b*: No such file or directory
$ shopt -s nullglob
$ for filename in a* b*
> do
>   md5sum $filename
> done
674ea002ddbaf89619e280f7ed15560d abc
1d48a9f8e8a42b0977ec8746cd484723 abcdef
b7f0f386f706aelbc3c8fa3bffb0371c abcdefghijk
8116e5ba834943c9047b6d3045f45c8c abcdef.odt
ac100d51fbab3ca3677d59e63212cb32 abc.php
d41d8cd98f00b204e9800998ecf8427e abctxt
e45f6583e2a3feacf82d55b5c8ae0a60 abc.txt
a60b09767be1fb8d88cbb1afbb90fb9e abc.TXT
3df05469f6e76c3c5d084b41352fc80b alphabet
$

```

这是个让脚本输出保持美观而没有多余的错误消息的好方法。

类似的有 **failglob** 选项。这是处理文件名扩展与任何文件都不匹配问题的另一种方法。设置 **failglob** 选项意味着 shell 本身会把使用不匹配的表达式当成是 shell 错误，而不是像一般情况那样对命令进行处理。

```
$ shopt failglob
failglob          off
$ ls b*
ls: cannot access b*: No such file or directory
$ shopt -s failglob
$ ls b*
bash: no match: b*
$
```

关于文件名扩展，shell 最强大的选项是 `extglob`。它向 `bash` 提供 `ksh` 中已有的一些扩展的模式匹配特性。这些特性似乎都没有很好的文档说明，而且在编写本书时，`bash` 手册页中也没有对其语法的清晰描述。最近我遇到的一个问题涉及在 Linux 服务器上对大量的磁盘进行处理。最常见的磁盘设备驱动程序在 Linux 中名为 `sd`，且磁盘名为 `/dev/sda`、`/dev/sdb` 等。磁盘上的独立分区名为 `/dev/sda1`、`/dev/sda2` 等。列出系统中所有磁盘的分区似乎是一件很容易就能完成的任务：

```
for disk in /dev/sd?
do
    fdisk -l $disk
done
```

这样会获取实际磁盘设备本身——`/dev/sda` 和 `/dev/sdb`——而不是希望列出的磁盘中的分区 `/dev/sda1`、`/dev/sda2` 和 `/dev/sdb1`。然而，当 Linux 系统中磁盘数目超过 26 个（在 SAN[Storage Area Network] 中很常见）时，磁盘名称会循环回来，所以 `/dev/sdz` 之后的磁盘为 `/dev/sdaa`，接着是 `/dev/sdab` 等。在 `/dev/sdaz` 之后是 `/dev/sdba`、`/dev/sdbb` 和 `/dev/sdbc` 等。在这种情况下，简单的 `/dev/sd?` 模式就不够了。真正需要表达的是字母表示磁盘，字母后面跟上的数字表示分区。在本例中只需要表示磁盘本身，所以可以使用 `extglob`。所需的模式是 `/dev/sd` 后面跟上一个或多个 `[a-z]` 字符。这可以表示为 `+(a-z)`，也可以使用 `[:alpha:]` 代替 `(a-z)`，两种方式在本例中都有使用。



`extglob` 模式与普通 shell 文件名扩展模式非常相似，但是（和）在普通文件名扩展中是非法的。

```
shopt -s extglob
ls /dev/sd+([a-z])
for disk in /dev/sd+([[:alpha:]])
do
    fdisk -l $disk
done
```

我们还可以使用 `GLOBIGNORE` 变量来只查找磁盘而不是分区。用 `GLOBIGNORE` 设置成的模式会将文件名扩展结果中的匹配部分去掉。在第 3 章提到过这些，但此处我们用它来列出所有磁盘，而不包括它们的分区。分区与模式 `/dev/sd*[0-9]` 相匹配——也就是

/dev/sd 之后有任意字符，且以数字结尾。

```
GLOBIGNORE=/dev/sd*[0-9]
for disk in /dev/sd*
do
    fdisk -l $disk
done
```

有了 extglob，我们还能匹配提供一套完整的模式。整套 extglob 通配符如表 4-2 所示。模式列表是指由模式构成的列表，模式之间用管道符号(|)隔开，如(a|bc|d)。

表 4-2 通配符

模 式	匹 配
?(模式列表)	0 或 1 个模式
*(模式列表)	0 或多个模式
+(模式列表)	1 或多个模式
@(模式列表)	1 个模式
!(模式列表)	除了一个模式以外的任何模式

问号用来可选地对模式列表进行标记。

```
$ shopt -s extglob
$ ls abc*
abc abcdef abcdefghijk abcdef.odt abc.php abctxt abc.txt abc.TXT
$ ls abc?(.)txt
abctxt abc.txt
$ ls abc?(def)
abc abcdef
$ ls abc?(def|.txt)
abc abcdef abc.txt
$
```

星号匹配模式出现 0 次或多次的情况，所以这些模式可以匹配选择性扩展，而不匹配所有 abc*。

```
$ shopt -s extglob
$ ls abc*
abc abcdef abcdefghijk abcdef.odt abc.php abctxt abc.txt abc.TXT
$ ls abc*(.php)
abc abc.php
$ ls abc*(.php|.txt)
abc abc.php abc.txt
$
```

加号要求至少有 1 个匹配存在。与上一段的星号相比，文件 abc 不足以匹配加号模式。

```
$ ls abc*(.txt|.php)
abc abc.php abc.txt
$ ls abc+ (.txt|.php)
abc.php abc.txt
$
```

@符号匹配模式出现一次的情况。创建 abc.txt.txt 文件来比较它与其他已测试过的形式之间的区别。

```
$ ls abc@(.txt|.php)
abc.php abc.txt
$ touch abc.txt.txt
$ ls abc@(.txt|.php)
abc.php abc.txt
$ ls abc+ (.txt|.php)
abc.php abc.txt abc.txt.txt
$ ls abc*(.txt|.php)
abc abc.php abc.txt abc.txt.txt
$
```

最后一个 extglob 符号是感叹号。它表示不匹配列表中的模式，刚好与@相反。

```
$ ls abc*
abc abcdefghijk abc.php abc.txt abc.txt.txt
abcdef abcdef.odt abctxt abc.TXT
$ ls abc@(.txt|.php)
abc.php abc.txt
$ ls abc! (.txt|.php)
abc abcdef abcdefghijk abcdef.odt abctxt abc.TXT abc.txt.txt
$
```

在 Unix 与 Linux 系统中，文件名区分大小写。也就是说，我们可以创建/home/steve/mydoc、/home/steve/MyDoc 和/home/steve/MYDOC，且它们各自是不同的文件。因为对于文件名而言大小写意义重大，所以文件名扩展一般也区分大小写。然而，当处理非原生文件系统时，如 VFAT 文件系统，不区分大小写可能比较有用。shell 选项 nocaseglob 可以启用这一特性。

```
$ shopt nocaseglob
nocaseglob      off
$ ls -ld /windows/program*
ls: cannot access /windows/program*: No such file or directory
$ shopt -s nocaseglob
$ ls -ld /windows/program*
drwxrwxrwx 1 root root 8192 Oct 28 19:06 /windows/ProgramData
drwxrwxrwx 1 root root 8192 Jun 11 2010 /windows/Program Files
drwxrwxrwx 1 root root 12288 Oct 28 19:04 /windows/Program Files (x86)
$
```

4.2 正则表达式和引用

正则表达式与 bash 通配符扩展的区别在于前者要远比后者完备，定义也比后者更清

晰。关于正则表达式可以写出(并且已经有)整本整本的书。它们与 `shell` 没有直接关系, 因为除了 `bash` 的 `=~` 语法, 只有像 `grep`、`sed` 和 `awk` 这样的外部工具才使用正则表达式。因为 `shell` 扩展与正则表达式使用的语法非常相似, 所以从 `shell` 传递给外部命令的正则表达式可能会在中途被 `shell` 分析。这一问题可以使用各种引用技术来避免。

4.2.1 正则表达式概述

正则表达式由 `sed` 这样的命令解释来实现某些非常强大的功能。虽然这些命令在本书的各种实用脚本中都有使用, 但它们的实际用法均超出本书的讨论范围。网站 <http://sed.sourceforge.net/sed1line.txt> 上提供了一系列极好的单行 `sed` 实用脚本, 另外还有很多在线教程与教材详细介绍了这些命令。此处简要介绍一些细节, 以方便讨论它们在 `shell` 中的用法。

```
$ cat myfile
foo="hello is bonjour"
bar="goodbye is aureviour"
fool=$foo
bar1=$bar
$ foo=bonjour
$ bar=aurevoir
$ sed s/$foo/$bar/g myfile
foo="hello is aurevoir"
bar="goodbye is aureviour"
fool=$foo
bar1=$bar
$ sed s/"$foo"/"$bar"/g myfile
foo="hello is aurevoir"
bar="goodbye is aureviour"
fool=$foo
bar1=$bar
$
```

在这个简单的例子中, 传递给 `sed` 的是 `s/bonjour/aurevoir/g`, 它将输入中的 `bonjour` 替换成 `aurevoir`。变量周围使用双引号也是一样的效果。然而, 如果在变量引用两旁使用单引号, 就不是按照之前的方式进行解释了。实际上, 字符串字面值 `$foo` 会被替换成 `$bar`, 而不会使用变量在调用 `shell` 中的值。传递给 `sed` 的指令是 `s/$foo/$bar/g`, 且 `sed` 不知道关于调用 `shell` 中变量的任何信息。



`sed` 是一种流编辑器。它不会修改磁盘中文件的内容, 只会读取文件(或标准输入), 然后将修改后的内容写到标准输出。GNU `sed` 使用 `-i` 选项直接对文件进行更新。

```
$ sed s/'$foo'/'$bar'/g myfile
foo="hello is bonjour"
bar="goodbye is aurevoir"
```

```
fool=$bar
barl=$bar
$
```

这就是说，我们应当理解不同引用的工作方式，以及依据不同工作方式决定应当向 `sed` 传递什么样的指令。引用的规则有时相当复杂，且不会总是很明显。

4.2.2 引用

从 `shell` 向外部命令传递参数的方式非常重要。有 3 种主要形式的引用——单引号、双引号以及反斜线。它们各自功能不同，尽管很多情况下其中某一两个就够用，但有时还是需要使用某个其他引用无法提供的特性。

1. 单引号

最简单的是单引号，它可以防止 `shell` 解释其中的内容。`shell` 分析的只是与第一个单引号配对的下一个单引号。

```
$ echo 'hello' 'world'
Helloworld
$ echo 'hello      world'
hello      world
$ echo '$hello'$world
$hello
$ echo 'hello
> world'
hello
world
$ echo *
cc2.ods CH3_PE.docx keyring-VH3EQr MozillaMailnews orbit-root orbit-steve
plugtmp pulse-ca5EDFdkeDRj ssh-aRFHoS1883 virtual-steve.VQ1hrC
$ echo '*'
*
```

第一个例子说明引号自身被忽略，但除了引号，`hello` 和 `world` 被当成完整的输入。第二个例子说明引号中的空格被保留。第三个例子说明 `$hello` 被当成字面字符串，但是未定义的变量 `$world` (不在引号之中) 被显示为空。第四个例子说明即使是换行符也不能终止 `echo` 语句——`shell` 知道还要等待一个单引号来结束引用，且不会对任何其他字符特殊对待，包括换行符。

最后两个例子说明尽管 `echo *` 扩展了当前目录中的所有文件，但是 `echo '*'` 只回显一个星号。

2. 双引号

`shell` 中第二种引用形式是双引号。用双引号引用，有部分字符会被 `shell` 解释，而不是全部。变量被解释，但是正如本章前半部分介绍的，文件名不会被扩展。这里给出的例子与上一节相同，但使用的是双引号。结果显示，尽管两种形式看似一样，但它们之间存

在较大的区别。

```
$ echo "hello""world"
helloworld
$ echo "hello      world"
hello      world
$ echo "$hello"$world
$ echo "hello
> world"
hello
world
$ echo *
cc2.ods CH3_PE.docx keyring-VH3EQr MozillaMailnews orbit-root orbit-steve
plugtmp pulse-ca5EDFdkeDRj ssh-aRFHoS1883 virtual-steve.VQ1hrC
$ echo "*"
*
$
```

双引号中的单引号也被保留。使用单引号与双引号分别引用对方来说明它们独立的作用方式。单引号中，所有字符按字面意义进行处理，包括双引号。双引号中，单引号被当成常规字符。

```
$ echo 'hello "world"'
hello "world"
$ echo "hello 'world'"
hello 'world'
$
```

常见的错误发生在将单引号用于常规文本中。如果需要显示一条如 `Let's play a game` 这样的消息，句子中的单个单引号可能导致一些问题。在下面的例子中，前两次 `echo` 尝试都需要一个用来结尾的单引号来结束整个命令，而且它们都没有能显示出目标文本。正确的方法是使用双引号。这能确保单个单引号被当成常规字符，而不是对 `shell` 有特殊意义的字符。

```
$ echo Let's play a game
> '
Lets play a game

$ echo 'Let's play a game'
> '
Lets play a game

$ echo "Let's play a game"
Let's play a game
$
```

3. 反斜线

第三种标记特殊字符的方法是单独用反斜线(`\`)作为它们的前缀。当需要在常规字符串

中包含特殊字符，但它又会被 shell 解释的时候，我们可以在字符前加上反斜线。在下面的例子中，can't 中的单引号本应当不出现这样的问题，因为它被双引号包围。可惜，事实并非如此。

```
$ echo "Wilde said, "Experience is one thing you can't get for nothing.""
> '
Wilde said Experience is one thing you cant get for nothing."
```

实际上，shell 会将其解释成下列几个不同的字符串。

- Wilde said, "
- Experience is one thing you can
- 't get for nothing.
- ""

如果要正确显示，我们需要让 shell 忽略形成引用的引号。在双引号前加上一个反斜线可以使引号失去其特殊意义。

```
$ echo "Wilde said, \"Experience is one thing you can't get for nothing.\""
Wilde said, "Experience is one thing you can't get for nothing."
$
```

另外一些可以使用反斜线进行转义的字符有：分号(;), 一般被 shell 用于在一行内组合多条命令；感叹号(!), 用于回调历史命令；&符号，用于在后台运行进程。在花括号扩展中，特殊符号{、}和,可以通过反斜线转义得到。另外，还可以对命令行中的反斜线本身进行转义(\\)。

```
$ echo the semicolon is wonderful; it is like taking a short break.
the semicolon is wonderful
-bash: it: command not found
$ echo the semicolon is wonderful\; it is like taking a short break.
the semicolon is wonderful; it is like taking a short break.
$
$ echo hope, peace & love
[1] 7517
hope, peace
-bash: love: command not found
[1]+ Done                  echo hope, peace
$ echo hope, peace \& love
hope, peace & love
$
$ echo hello
hello
$ echo !echo
echo echo hello
echo hello
$ echo \!echo
!echo
```



```
$
$ echo DOS and Windows refer to drives as C:\, D:\ and E:\.
DOS and Windows refer to drives as C:, D: and E:.
$ echo DOS and Windows refer to drives as C:\\, D:\\ and E:\\.
DOS and Windows refer to drives as C:\, D:\ and E:\.
$
```



在双引号中，`$`、```、`\$`、`\'`、`\"`以及`\<newline>`都被当成普通字符。反斜线与其他任何字符的组合将不被特殊处理。所以 `echo "$HOME\&"` 显示 `/home/steve\&`，而不是在不使用双引号时的 `/home/steve&`。

有些命令需要特殊字符作为它们语法的一部分。例如，`sed` 命令一般会包含正斜线。如果要将正斜线作为 `sed` 命令的一部分而不是语法的一部分，则需要对 `sed` 中的正斜线进行转义。下面的例子需要用到引号(单引号或者双引号)与反斜线。

```
$ cat files.txt
/etc/hostnames
$ sed s/hostnames/hosts/g files.txt
/etc/hosts
$ sed s//etc/hostnames//etc/hosts/g files.txt
sed: -e expression #1, char 8: unknown option to `s'
$ sed s"/etc/hostnames"/etc/hosts/g files.txt
sed: -e expression #1, char 8: unknown option to `s'
$ sed s\/etc\/hostnames\/etc\/hosts/g files.txt
sed: -e expression #1, char 8: unknown option to `s'
$ sed s"/etc\/hostnames"/etc\/hosts/g files.txt
/etc/hosts
$
```



这就是 `sed` 语法，不是一般的 shell 语法。如果要对其进行演示，请用 `echo` 代替 `sed` 来重复以上的测试命令。`echo` 并不关心正斜线，它把正斜线当成常规字符。

反斜线还能用于续行。一个很长的命令会很难读懂，所以最好选一个位置进行换行。如果一行代码以反斜线结尾，并且紧跟反斜线之后的是换行符，则在解释时反斜线与换行符都会被去掉。这适用于 shell 脚本与命令提示符。

```
$ cat cont.sh
#!/bin/bash
echo foo\
bar
$ ./cont.sh
foobar
$
```

在向外部程序传递变量的值时，这些变量解释的顺序对于程序运行结果有很大影响。首先看第一例，命令 `ssh declan echo $PATH` 被服务器 `goldie` 中的 `shell` 解释为 `ssh declan echo /home/steve/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games`，因为那就是 `goldie` 中 `PATH` 变量的值。为确保字符串 `$PATH` 不被 `goldie` 中的 `shell` 解释，我们用反斜线作为前缀，于是 `goldie` 中的 `shell` 看不见 `$PATH`，也无法替换成它的值；但是服务器 `declan` 中的 `shell` 能看见 `$PATH`，并用 `declan` 中的 `$PATH` 值对它进行正确的扩展。

```
steve@goldie:~$ echo $PATH
/home/steve/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games

steve@declan:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/steve/bin
steve@declan:~$

steve@goldie:~$ ssh declan echo $PATH
steve@declan's password:
/home/steve/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games

steve@goldie:~$ ssh declan echo \ $PATH
steve@declan's password:
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/steve/bin
steve@goldie:~$
```

4.3 本章小结

通配符非常灵活且有用，但是为此付出的代价是使用的时候必须特别小心。一旦使用到任何非标准字符，我们就需要考虑 `shell` 对它们的处理方法。某些情况下可以直接通过，另外一些情况下这些字符会被认为含有特殊意义。这些不同的情况并不总是很好区分——规则比较复杂，而且不是很容易理解。

对于用户产生的或来自外部输入的字符串的处理方式，我们应当做到了然于胸。例如，如果所有的测试都假设用户提供的文件名不包含空格，一旦用户传递某个包含空格的文件名且没有将文件名引用起来，那么脚本就会崩溃。

第5章介绍条件执行。使用条件执行对环境状态进行测试，以便控制 `shell` 脚本的运行流程。

第 5 章

条 件 执 行

条件执行是指如果满足某些条件才执行代码。如果没有条件执行，则只能逐条执行命令。如果 shell 脚本能对系统状态以及进程的环境变量进行测试，则能实现更强大的功能。

5.1 if/then

几乎每种编程语言都有 if/then/else 结构，shell 也不例外。shell 语法使用方括号来进行测试，并使用 then 和 fi 语句，作用与 C 以及其他一些语言中的 {和} 一样。

```
if [ condition ]
then
    statement(s)
fi
```

除了 then 之后的换行符，所有换行都是必需的，或者使用分号代替。[和]中的空格也是必需的。所以，上面的代码可以压缩为：

```
if [ condition ];then statement(s);fi
```

经常使用分号将 then 与 if 放置在一行中。使用单词反转来表示其相反的意思——如使用 fi 来终止 if 语句——本章后面还会介绍这种用法，如使用 esac 终止 case 语句。类似的如第 12 章将介绍的，cat 的反转为 tac。



Unix、Linux 和自由软件传统使用了很多的双关语。除了第 1 章提到的 GNU 和 HURD 中使用的花样外，Perlmongers 发起了 Yet Another Perl Conference (YAPC)，SuSE 引入了 Yet Another Software Tool (YaST)，abcde 表示 A Better CD Encoder。拉丁美洲的 GNU National User Groups 被称为 GNU^2。这样的例子不胜枚举。

在上面的代码中添加条件不满足时运行的代码段，这样可以让它更实用。

```
if [ condition ]
then
    statements for when the condition is true
else
    statements for when the condition is false
fi
```

在上面的代码中，如果条件测试通过，则执行第一部分的语句，反之则执行第二部分的语句。只有其中一个代码块会被执行——绝不会两个都执行，也不会都不执行。

通过检查\$?返回变量可以测试运行成功与否。\$?为0表示成功，否则表示失败。在这些测试中，我们使用-r测试，它返回真(0)的条件是当且仅当文件存在且可读。下面的代码试图显示以第一个参数传递过去的文件(用双引号括起来可以允许文件名包含空格)，如果失败则给出错误消息(cat也会向stdout输出本身运行的错误消息)。

```
#!/bin/bash
# Test for failure
cat "$1"
if [ "$?" -ne "0" ]; then
    echo "Error: Reading $1 failed."
fi
```

下面的代码在试图访问文件之前查看文件是否存在且可读。这样做更安全；更好一点的做法是在cat命令运行之后检查是否有未预见的失败条件(如cat可能不在\$PATH中)。

```
#!/bin/bash
# Test for likely causes of failure

if [ ! -r "$1" ]; then
    echo "Error: $1 is not a readable file."
    echo "Quitting."
    exit 1
fi

cat "$1"
```

5.2 else

我们希望尽可能对文件运行cat命令，但是如果无法运行，则继续执行脚本。一种方法是使用两个测试。下面是第一个测试：

```
if [ -r "$1" ]; then cat "$1"; fi
```

紧接着是相反的测试(!将测试结果取反)：

```
if [ ! -r "$1" ]; then echo "File $1 is not readable - skipping. "; fi
```

但是这很麻烦且容易出错。如果以后要将测试替换为-s, 则必须对两处代码进行修改。所以, else 语句应运而生:

```
#!/bin/bash
# Check for likely causes of failure

if [ ! -r "$1" ]; then
    echo "Error: $1 is not a readable file."
else
    cat "$1"
fi
```

将测试中的感叹号去掉, 并交换两条语句的次序, 这样代码更易读:

```
#!/bin/bash
# Check for likely causes of failure

if [ -r "$1" ]; then
    cat "$1"
else
    echo "Error: $1 is not a readable file."
fi
```

这样更具可读性, 且比之前的脚本更具健壮性。简明的方案通常比更复杂的方案易懂。

5.3 elif

elif 结构允许为 if 语句的 else 部分添加条件测试。它是 else if 的缩写, 以更简明的写法来表示 else 与 if 这一连串行为。elif 易写、易读, 最重要的是便于调试。跨平台脚本(如各种不同 Unix 下的通用安装程序)的常见任务是根据脚本实际运行的操作系统来执行任务的不同部分。如果连一点与平台相关的代码都没有, 脚本阅读、编辑与调试起来显然会比较困难。



可从

wrox.com
下载源代码

```
#!/bin/bash
OS=`uname -s`

if [ "$OS" = "FreeBSD" ]; then
    echo "This Is FreeBSD"
else
    if [ "$OS" = "CYGWIN_NT-5.1" ]; then
        echo "This is Cygwin"
    else
        if [ "$OS" = "SunOS" ]; then
            echo "This is Solaris"
        else
```

```

if [ "$OS" = "Darwin" ]; then
    echo "This is Mac OSX"
else
    if [ "$OS" = "AIX" ]; then
        echo "This is AIX"
    else
        if [ "$OS" = "Minix" ]; then
            echo "This is Minix"
        else
            if [ "$OS" = "Linux" ]; then
                echo "This is Linux"
            else
                echo "Failed to identify this OS"
            fi
        fi
    fi
fi
fi
fi
fi
fi

```

elif1.sh

`elif` 可以让代码更加简单，不仅增强可读性，并且让脚本处于易于维护的状态。如果要向 `elif1.sh` 中添加另一个操作系统，我们必须计算好代码缩进，否则(实际使用时就会遇到这样的情况)缩进会变得一团糟，导致整块代码无法阅读。所以，下一个用户添加新的操作系统的任务将更加困难。



可从

wrox.com
下载源代码

```

#!/bin/bash
OS=`uname -s`

if [ "$OS" = "FreeBSD" ]; then
    echo "This Is FreeBSD"
elif [ "$OS" = "CYGWIN_NT-5.1" ]; then
    echo "This is Cygwin"
elif [ "$OS" = "SunOS" ]; then
    echo "This is Solaris"
elif [ "$OS" = "Darwin" ]; then
    echo "This is Mac OSX"
elif [ "$OS" = "AIX" ]; then
    echo "This is AIX"
elif [ "$OS" = "Minix" ]; then
    echo "This is Minix"
elif [ "$OS" = "Linux" ]; then
    echo "This is Linux"
else
    echo "Failed to identify this OS"
fi

```

elif2.sh

要想在上面的代码中添加新的操作系统，我们只要在脚本中再添加两行代码。这不会涉及代码缩进与可读性的问题，而且代码含义也是非常清晰的。

5.4 test([])

本章的第一例使用 `-r` 测试来检查文件是否存在且可读。对于这个例子而言，`-r` 是一个很有用的测试，但是 `test` 手册页中介绍了更多附有详细文档的测试。这里没有必要将整个 `test` 手册页重复一遍。查看一些最有用的测试选项，并将它们适当地运用于手头的任务。由于种类太多，很容易看见一些 `shell` 脚本用其他方法重复实现了一些测试功能，这仅仅是因为脚本作者并不知晓测试功能的强大。

首先要承认的是，测试的实现方式很有技巧性。这种实现方式还可以解释本章开头提到的有关空格的规则。`test` 本身是一个程序，通常作为 `shell` 的内置命令。所以尽管磁盘上一般都会有 `/usr/bin/test`，但它不会被调用，因为首先找到的是 `shell` 的内置命令。`test` 的另一个名称为 `[]`。当调用 `[]` 时，它需要一个 `[]` 作为其参数，但如果不使用 `[]`，`[]` 也能起到同样的作用。

```
$ type test
test is a shell builtin
$ type [
[ is a shell builtin
$ which test
/usr/bin/test
$ which [
/usr/bin/[
$ ls -il /usr/bin/test /usr/bin/[
33625 -rwxr-xr-x 1 root root 33064 Apr 28 2010 /usr/bin/[
33634 -rwxr-xr-x 1 root root 30136 Apr 28 2010 /usr/bin/test
```

本例说明，如果在 `bash` 中调用 `test` 或 `[]`，则会使用它们在 `bash` 中的实现。磁盘中也会有大小相近的 `test` 与 `[]`，但它们是不同的文件。

有些计算机语言具有比 `shell` 简单得多的分析器，而 `shell` 的这种显得有些奇怪的配置使 `shell` 表现得像与这些语言具有相同的语法。

最后的结果是，对于习惯于其他语言的程序员而言，尽管 `shell` 代码看起来整齐而具有可识别性，但与其他语言的处理方式存在巨大差异。在大多数语言中，`[`和`]`(或它们的等价符号)是语言的一部分。在 `shell` 中，`[]`是程序，而`[]`只是`[]`所需的参数。但如果不使用`[]`，也不会对 `shell` 有任何影响。

也就是说，我们可以用多种方法测试 `/etc/hosts` 是否为常规文件：

```
$ test -f /etc/hosts
$ echo $?
0
$ /usr/bin/test -f /etc/hosts
$ echo $?
```



```
0
$ [ -f /etc/hosts ]
$ echo $?
0
```

本例使用了 3 个不同的程序，它们都执行相同的测试，且都返回 0(表示测试成功)。我们还可以测试/etc/stsoh 不是常规文件：

```
$ test -f /etc/stsoh
$ echo $?
1
$ /usr/bin/test -f /etc/stsoh
$ echo $?
1
$ [ -f /etc/stsoh ]
$ echo $?
1
```

本例中，3 个程序结果相同，返回码都为 1(只要是任意非零值就行)。了解代码背后的行为有利于对测试的理解。

5.4.1 测试标志

要测试文件是否存在，我们使用 -e 标志(很容易记住，因为 e 表示 exists)。-a 标志与 -e 含义相同。

```
if [ -e /etc/resolv.conf ]; then
    echo "DNS Configuration:"
    cat /etc/resolv.conf
else
    echo "No DNS resolv.conf file exists."
fi
```

因为 Unix 下一切皆文件，所以上面的测试看上去比较特别，但实际上并非如此。设备驱动程序、目录、网络挂载以及虚拟内存空间都可以表示为文件。一个与 -e 类似的测试是 -f，它测试文件是否存在且为常规文件。另外，-b 测试文件是否为块设备，-c 测试文件是否为字符设备。块设备的驱动对象是像硬盘这样的按块进行操作的设备。字符设备的驱动对象可以向其读写字符的设备，如终端或者/dev/random 虚拟设备。

文件可以是硬链接或者符号链接。硬链接实际上是文件本身，而符号链接(指向可能处于不同文件系统上的实际文件的指针)要使用 -L 标志进行测试，或者也能使用 -h 标志。常规文件的链接也能通过 -f 测试，块设备的链接能通过 -b 测试等。下面的代码在 -f 测试之前使用 -L 测试。否则，脚本将认为测试对象为常规文件而不会测试它是否为符号链接。

Linux 中的文件还可能是套接字(使用 -S 标志进行测试)与命名(FIFO)管道(使用 -p 标志进行测试)。套接字是进程间通信的更高级特性，不在本书的讨论范围之内。第 14 章将介绍管道。

下面这个脚本测试所有这些可能性:



可从
wrox.com
下载源代码

```
$ cat blockcharacterfile.sh
#!/bin/bash
while read -p "What file do you want to test? " filename
do
    if [ ! -e "$filename" ]; then
        echo "The file does not exist."
        continue
    fi

    # Okay, the file exists.
    ls -ld "$filename"

    if [ -L "$filename" ]; then
        echo "$filename is a symbolic link"
    elif [ -f "$filename" ]; then
        echo "$filename is a regular file."
    elif [ -b "$filename" ]; then
        echo "$filename is a block device"
    elif [ -c "$filename" ]; then
        echo "$filename is a character device"
    elif [ -d "$filename" ]; then
        echo "$filename is a directory"
    elif [ -p "$filename" ]; then
        echo "$filename is a named pipe"
    elif [ -S "$filename" ]; then
        echo "$filename is a socket"
    else
        echo "I don't know what kind of file that is. Is this a Linux system?"
    fi
done
$
$ ./blockcharacterfile.sh
What file do you want to test? /etc/foo
The file does not exist.
What file do you want to test? /etc/hosts
-rw-r--r-- 1 root root 458 Dec 3 00:23 /etc/hosts
/etc/hosts is a regular file.
What file do you want to test? /dev/sda
brw-rw---- 1 root disk 8, 0 Dec 13 09:22 /dev/sda
/dev/sda is a block device
What file do you want to test? /dev/null
crw-rw-rw- 1 root root 1, 3 Dec 13 09:22 /dev/null
/dev/null is a character device
What file do you want to test? /etc
drwxr-xr-x 141 root root 12288 Dec 13 09:24 /etc
/etc is a directory
What file do you want to test? /etc/motd
lrwxrwxrwx 1 root root 13 Jun 5 2010 /etc/motd -> /var/run/motd
```

```

/etc/motd is a symbolic link
What file do you want to test? /tmp/OSL_PIPE_1000_SingleOfficeIPC
_54f1d8557767a73f9 bc36a8c3028b0
srwxrwxr-x 1 steve steve 0 Dec 13 10:32 /tmp/OSL_PIPE_1000
_SingleOfficeIPC_54f1d855 7767a73f9bc36a8c3028b0
/tmp/OSL_PIPE_1000_SingleOfficeIPC_54f1d8557767a73f9bc36a8c3028b0 is a socket
What file do you want to test? /tmp/myfifo
prwx----- 1 steve steve 0 Dec 14 12:47 /tmp/myfifo
/tmp/myfifo is a named pipe
What file do you want to test? ^C
$

```

blockcharacterfile.sh

另一个使用-d 标志的是/etc/profile 脚本。该脚本能在当前系统状态下对用户环境进行自定义。下面的代码将~/bin 添加到 PATH 环境变量中,但只有当该目录存在的时候才添加:

```

if [ -d ~/bin ]; then
    PATH=$PATH:~/bin
fi

```

除了类型以外,文件还具有很多其他属性,包括大小、权限与时间戳等。Unix 文件权限存储在 3 个独立块中,每个文件都有一个属主与属组(属主与属组通常会重叠,但也不总是如此)。每个权限块都定义了读(R)、写(W)和执行(X)位。一般表示为 **rwX**,如果没有相应权限则标记为短横线(-)。第一个字符表示文件类型,也就是之前代码测试的文件类型。然后是 9 个表示文件权限的字符。第一个权限块决定了文件属主的权限;第二个块决定了文件属组成员的权限;第三个块决定了属主与属组以外其他任何人具有的权限。例如,下面的配置文件的属主是 **root**,并可以被 **root** 读或写(第一个权限块 **rw-**)。它的属组是 **fuse** 组,该组的成员只能读取该文件(**r--**)。最后 3 个短横线(---)表示其他任何人对该文件都不具有任何权限。

```
-rw-r----- 1 root fuse 216 Jan 31 2010 /etc/fuse.conf
```

尽管刚开始很容易会认为有一个 **fuse** 用户拥有该文件,这样应用程序可以从文件中读取相关配置,但是配置只能被 **root** 修改,防止 **fuse** 对其修改。这本身不是什么安全措施,但却是个不错的深度防护策略。Unix 与 Linux 用来抵御意外攻击的许多方法中就包括深度防护策略。

前 3 个测试是 **-r**、**-w** 和 **-x**。这些测试检测该特定会话对测试文件是否具有读、写和执行权限。对目录具有执行权限表示可以进入该目录。这里要注意的是,并不是直接对文件权限进行测试,来查询进程可以对文件进行何种操作。同一个脚本对同一个文件进行操作也可能有不同的输出,这取决于运行用户而不是脚本操作的目标文件。另外,文件的权限可能是 **rwXrwXrwX**,但如果用户没有文件所在目录的权限,则文件也是完全无法访问的。



可从
wrox.com
下载源代码

```
$ cat rwx.sh
#!/bin/bash
while read -p "What file do you want to test? " filename
do
    if [ ! -e "$filename" ]; then
        echo "The file does not exist."
        continue
    fi

    # Okay, the file exists.
    ls -ld "$filename"
    if [ -r "$filename" ]; then
        echo "$filename is readable."
    fi
    if [ -w "$filename" ]; then
        echo "$filename is writeable"
    fi
    if [ -x "$filename" ]; then
        echo "$filename is executable"
    fi
done
$ ./rwx.sh
What file do you want to test? /home/steve
drwxr-xr-x 70 steve steve 4096 Dec 13 11:52 /home/steve
/home/steve is readable.
/home/steve is writeable
/home/steve is executable
What file do you want to test? /etc
drwxr-xr-x 141 root root 12288 Dec 13 11:40 /etc
/etc is readable.
/etc is executable
What file do you want to test? /etc/hosts
-rw-r--r-- 1 root root 458 Dec 3 00:23 /etc/hosts
/etc/hosts is readable.
What file do you want to test? /etc/shadow
-rw-r----- 1 root shadow 1038 Nov 4 18:32 /etc/shadow
What file do you want to test? ^C
$
```

rwx.sh

我们还可以使用-O和-G标志分别检测文件是否属于当前用户和/或组ID。这比起找出用户ID或组ID以及与文件相关的相同信息，再比较两者是否相等的方法要简单得多。



可从
wrox.com
下载源代码

```
$ cat owner.sh
#!/bin/sh
while read -p "What file do you want to test? " filename
do
    if [ ! -e "$filename" ]; then
        echo "The file does not exist."
        continue
    fi
```

```

fi

# Okay, the file exists.
ls -ld $filename
if [ -O $filename ]; then
    echo "You own $filename"
else
    echo "You don't own $filename"
fi
if [ -G $filename ]; then
    echo "Your group owns $filename"
else
    echo "Your group doesn't own $filename"
fi
done
$ ./owner.sh
What file do you want to test? /home/steve
drwxr-xr-x 70 steve steve 4096 Dec 13 23:24 /home/steve
You own /home/steve
Your group owns /home/steve
What file do you want to test? /etc/hosts
-rw-r--r-- 1 root root 458 Dec 3 00:23 /etc/hosts
You don't own /etc/hosts
Your group doesn't own /etc/hosts
What file do you want to test? ^C
$

```

owner.sh

Unix 风格的文件权限的另一个特性是 `suid`(Set UserID)位与 `sgid`(Set GroupID)位。它们允许程序以文件所属用户(或组)的身份运行，而不一定是运行程序的用户(或组)的身份。这在 `rwX` 模式中使用 `s` 而不是 `x` 来表示文件权限。我们可以分别使用 `-g` 和 `-u` 来测试这两个权限。



可从
wrox.com
下载源代码

```

$ cat suid.sh
#!/bin/sh
while read -p "What file do you want to test? " filename
do
    if { ! -e "$filename" }; then
        echo "The file does not exist."
        continue
    fi
fi

# Okay, the file exists.
ls -ld $filename
if [ -u $filename ]; then
    echo "$filename will run as user \"`stat --printf=%U $filename`\""
fi
if [ -g $filename ]; then
    echo "$filename will run as group \"`stat --printf=%G $filename`\""

```

```

fi
done
$ ./suid.sh
What file do you want to test? /usr/bin/procmail
-rwsr-sr-x 1 root mail 89720 Apr 25 2010 /usr/bin/procmail
/usr/bin/procmail will run as user "root"
/usr/bin/procmail will run as group "mail"
What file do you want to test? /bin/ping
-rwsr-xr-x 1 root root 34248 Oct 14 07:21 /bin/ping
/bin/ping will run as user "root"
What file do you want to test? ^C
$

```

suid.sh

在有些情况下，如查看日志文件是否已写入，检查文件是否含有任何内容非常有用。`-s` 标志能测试文件是否存在且不为空。如果通过`-s` 测试，则文件不为空。如果失败，文件可能不存在或者大小为 0。取决于所需知道的组合方式，`-s` 可能已经够用，但可以将其与`-e` 组合来检查文件是否存在。在一些系统中，`/var/log/mcelog` 包含所有检测到的机器检查异常的日志。我们希望没有任何异常，但可以使用简单的脚本来检测是否存在问题。下面的脚本发现`/var/log/mcelog` 中有 18 行内容：



```

$ cat mce.sh
#!/bin/sh
if [ -s /var/log/mcelog ]; then
    echo "Machine Check Exceptions found : "
    wc -l /var/log/mcelog
fi
$ ./mce.sh
Machine Check Exceptions found :
18 /var/log/mcelog
$

```

mce.sh

有时候需要在文件包含新内容时读取它。命名管道(也称为先入先出或者 FIFO)可能是个较好的解决方案，但有时却无法选择数据的生成方式。`-N` 标志检测文件自从上次读取之后是否被修改。可以使用两个脚本对`-N` 测试的功能进行演示——图形环境下可以在两个不同窗口中运行脚本，或者使用连接到服务器的两个独立会话。



```

$ echo hello > /tmp/myfile.log
$ echo hello world >> /tmp/myfile.log
$ cat watchfile.sh
#!/bin/bash
GAP=10                                # how long to wait
LOGFILE=$1                            # file to log to

# Get the current length of the file.
len=`wc -l $LOGFILE | awk '{ print $1 }'`

```

```
echo "Current size is $len lines"

while :
do
    if [ -N $LOGFILE ]; then
        echo "`date`: New entries in $LOGFILE:"
        newlen=`wc -l $LOGFILE | awk '{ print $1 }'`
        newlines=`expr $newlen - $len`
        tail -$newlines $LOGFILE
        len=$newlen
    fi
    sleep $GAP
done
$ ./watchfile.sh /tmp/myfile.log
Current size is 2 lines
```

watchfile.sh

现在从一个独立的窗口中运行下面的命令:

```
$ echo this is a test >> /tmp/myfile.log
```

在第一个窗口中, 我们将在\$GAP 秒(本例中是 10 秒)内看到下面的输出:

```
Fri Dec 17 10:56:42 GMT 2010: New entries in /tmp/myfile.log:
this is a test
```

然后在第二个窗口中运行下面的命令:

```
$ echo this is a two line test >> /tmp/myfile.log ; \
> echo this is the second line of it >> /tmp/myfile.log
$
```

花多长时间输入上面那些命令都没有关系。只要当文件发生变化时, watchfile.sh 才显示数据, 所以在一行中使用两个 echo 命令同样能起作用。还可以增加\$GAP 的值(如 60 秒)来获取更多的输入时间。

第一个窗口将输出下面的内容:

```
Fri Dec 17 10:57:52 GMT 2010: New entries in /tmp/myfile.log:
this is a two line test
this is the second line of it
```

只要对日志文件本身执行 cat 命令就能确认文件中的内容:

```
$ cat /tmp/myfile.log
hello
hello world
this is a test
this is a two line test
this is the second line of it
```

这是一个查看文件增量变化的有效方法，甚至更适合于 cron 任务。

5.4.2 文件比较测试

test 命令还能对文件进行一些基本的比较操作。-ef 比较两个文件是否为同一文件系统中相同 inode 节点的硬链接。这省去很多麻烦，因为尽管 stat --format=%i 或者 ls -li 能提供文件的 inode 号，我们还是必须检查两个文件是否处于同一个文件系统中。



可从
wrox.com
下载源代码

```
$ cat equalfile.sh
#!/bin/bash

file1=$1
file2=$2

ls -li $file1 $file2
if [ $file1 -ef $file2 ]; then
    echo "$file1 is the same file as $file2"
else
    echo "$file1 is not the same file as $file2"
    diff -q $file1 $file2
    if [ "$?" -eq "0" ]; then
        echo "However, their contents are identical."
    fi
fi

$ echo testing > file1
$ ln file1 file2
$ echo testing > file3
$ ls -li file?
4931911 -rw-rw-r-- 2 steve steve 8 Dec 14 21:04 file1
4931911 -rw-rw-r-- 2 steve steve 8 Dec 14 21:04 file2
4931915 -rw-rw-r-- 1 steve steve 8 Dec 14 21:04 file3
$ ./equalfile.sh file1 file2
4931911 -rw-rw-r-- 2 steve steve 8 Dec 14 21:04 file1
4931911 -rw-rw-r-- 2 steve steve 8 Dec 14 21:04 file2
file1 is the same file as file2
$ ./equalfile.sh file1 file3
4931911 -rw-rw-r-- 2 steve steve 8 Dec 14 21:04 file1
4931915 -rw-rw-r-- 1 steve steve 8 Dec 14 21:04 file3
file1 is not the same file as file3
However, their contents are identical.
$ echo something different > file4
$ ./equalfile.sh file1 file4
4931911 -rw-rw-r-- 2 steve steve 8 Dec 14 21:04 file1
4931917 -rw-rw-r-- 1 steve steve 20 Dec 14 21:05 file4
file1 is not the same file as file4
Files file1 and file4 differ
$
```

equalfile.sh

test 能对两个文件执行的其他比较操作包括查看一个文件的内容的修改日期是否比另一个文件更晚。



只能比较修改时间(mtime), 不能对访问时间(ctime)与 inode 修改时间(ctime)进行测试。

```
$ echo old file > old
$ sleep 60
$ echo newer file > new
$ ls -l new old
-rw-rw-r-- 1 steve steve 11 Dec 14 13:21 new
-rw-rw-r-- 1 steve steve  9 Dec 14 13:20 old
$ if [ new -nt old ]; then
>   echo "new is newer"
> else
>   echo "old is newer"
> fi
new is newer
$ if [ new -ot old ]; then
>   echo "new is older"
> else
>   echo "old is older"
> fi
old is older
$
```



如果文件的时间戳相同, 上面两个测试都返回假, 因为没有哪个文件更新或更旧。同样, 如果某个文件不存在, 两个测试都返回真。如果两个文件都不存在, 两个测试都返回假。

5.4.3 字符串比较测试

有 4 种用于字符串的测试。可以测试字符串是否相等, 以及按照字母表测试某个字符串是否排在另一个之前。下面的脚本比较两个字符串并报告它们是否相等。如果不相等则报告哪个排在前面。要完成这一任务, 必须使用与目前所见的有些不一样的语法。字符串比较<和>只能在复合命令[[...]]中起作用。==和!=测试既可以在单方括号测试中也能在双方括号测试中使用, 但是单个等于符号(=)只能用在单方括号测试中。导致这种复杂性的原因是为了在添加更加强大的[[命令的同时, 保持与 Bourne shell 的兼容性。



可从
wrox.com
下载源代码

```
$ cat alnum.sh
#!/bin/bash
if [ "$1" = "$2" ]; then
    echo "$1 is the same as $2"
else
```

```

echo "$1 is not the same as $2"
# Since they are different, let's see which comes first:
if [[ "$1" < "$2" ]]; then
    echo "$1 comes before $2"
else
    echo "$1 comes after $2"
fi
fi
$ ./alnum.sh apples bananas
apples is not the same as bananas
apples comes before bananas
$ ./alnum.sh bananas apples
bananas is not the same as apples
bananas comes after apples
$ ./alnum.sh oranges oranges
oranges is the same as oranges
$

```

alnum.sh

alnum.sh 的第二行代码使用单个等于符号=来测试是否相等。bash 和其他 shell 也接受双等于符号==，这与其他一些语言(特别是 C 语言)更相似。然而，这与 POSIX 标准不兼容，并且传统的 Bourne shell 不承认这种语法。最好使用都能接受的单个等于符号=。

脚本中没有使用不相等的测试，应使用!=来实现：

```

$ if [ "one" != "two" ]; then
>   echo "These are not the same"
> fi
These are not the same
$

```

最后介绍的两个字符串测试与文件的-s 测试相似。-z 测试在字符串长度为 0 时返回真，而-n 在字符串长度非零时返回真。因为测试的字符串可能会是空字符串，所以需要引号将变量名引用起来；否则[-z \$input]就会成为[-z]，这从语法上而言是非法的。只有[-z ""]在语法意义上才是合法的。



可从
wrox.com
下载源代码

```

$ cat nz.sh
#!/bin/bash
# Set input to a known value as we are testing it before we set it
input=""

while [ -z "$input" ]; do
    read -p "Please give your input: " input
done
echo "Thank you for saying $input"

```

```
$ ./nz.sh
Please give your input:
Please give your input:
Please give your input: something
Thank you for saying something
$
```

`nz.sh`

5.4.4 正则表达式测试

bash 3 新增的一个特性是`==`操作符，它与 perl 中`==`符号的作用非常相似。它将右边当成扩展的正则表达式，所以以前需要使用 perl、sed 或 grep 才能完成的一些任务，现在 bash 就能完成。这意味着可以通过`[[$pkgname == *.deb]]`来查找匹配模式`*.deb`的文件名。注意要使用双方括号语法`[[...]]`。



可从
wrox.com
下载源代码

```
$ cat isdeb.sh
#!/bin/bash

for deb in pkgs/*
do
    pkgname=`basename $deb`
    if [[ $pkgname == *.deb ]]; then
        echo "$pkgname is a .deb package"
    else
        echo "File \"$pkgname\" is not a .deb package."
    fi
done

$ ls pkgs/
dbus-x11_1.2.24-4_amd64.deb
firmware-linux-free_2.6.32-29_all.deb
gnome-desktop-1.023.x86_64.rpm
libgnomeprint-2.18.6-2.6.x86_64.rpm
libgnomeui2-2.24.3-1mdv2010.1.i586.rpm
linux-headers-2.6.32-5-amd64_2.6.32-29_amd64.deb
linux-headers-2.6.32-5-common_2.6.32-29_amd64.deb
linux-libc-dev_2.6.32-29_amd64.deb
linux-source-2.6.32_2.6.32-29_all.deb
README

$ ./isdeb.sh
dbus-x11_1.2.24-4_amd64.deb is a .deb package
firmware-linux-free_2.6.32-29_all.deb is a .deb package
File "gnome-desktop-1.023.x86_64.rpm" is not a .deb package.
File "libgnomeprint-2.18.6-2.6.x86_64.rpm" is not a .deb package.
File "libgnomeui2-2.24.3-1mdv2010.1.i586.rpm" is not a .deb package.
linux-headers-2.6.32-5-amd64_2.6.32-29_amd64.deb is a .deb package
linux-headers-2.6.32-5-common_2.6.32-29_amd64.deb is a .deb package
linux-libc-dev_2.6.32-29_amd64.deb is a .deb package
linux-source-2.6.32_2.6.32-29_all.deb is a .deb package
File "README" is not a .deb package.
$
```

`isdeb.sh`

这个脚本很有用,但是它没有说明哪些是匹配的字符串。可以使用 `BASH_REMATCH[]` 数组来实现这一功能。任何由括号括起来的表达式都被放入了这个数组。索引号 0 匹配整个字符串。我们可以从该数组中获取精确的匹配项。根据 `.deb` 的命名约定(包名称_版本号_架构.deb), 可以从文件名中提取这些数据。



```
$ cat identify_debs.sh
#!/bin/bash

for deb in pkgs/*
do
    pkgname=`basename $deb`
    echo $pkgname
    if [[ $pkgname =~ (.)_(.*)_(.*)\.deb ]]; then
        echo "Package ${BASH_REMATCH[1]} Version ${BASH_REMATCH[2]}"\
            "is for the \"${BASH_REMATCH[3]}\" architecture."
        echo
    else
        echo "File \"$pkgname\" does not appear to match the "
        echo "standard .deb naming convention."
        echo
    fi
done

$ ./identify_debs.sh
dbus-x11_1.2.24-4_amd64.deb
Package dbus-x11 Version 1.2.24-4 is for the "amd64" architecture.

firmware-linux-free_2.6.32-29_all.deb
Package firmware-linux-free Version 2.6.32-29 is for the "all" architecture.

gnome-desktop-1.023.x86_64.rpm
File "gnome-desktop-1.023.x86_64.rpm" does not appear to match the
standard .deb naming convention.
libgnomeprint-2.18.6-2.6.x86_64.rpm
File "libgnomeprint-2.18.6-2.6.x86_64.rpm" does not appear to match the
standard .deb naming convention.

libgnomeui2-2.24.3-1mdv2010.1.i586.rpm
File "libgnomeui2-2.24.3-1mdv2010.1.i586.rpm" does not appear to match the
standard .deb naming convention.

linux-headers-2.6.32-5-amd64_2.6.32-29_amd64.deb
Package linux-headers-2.6.32-5-amd64 Version 2.6.32-29 is for the
"amd64" architecture.

linux-headers-2.6.32-5-common_2.6.32-29_amd64.deb
Package linux-headers-2.6.32-5-common Version 2.6.32-29 is for the
"amd64" architecture.

linux-libc-dev_2.6.32-29_amd64.deb
Package linux-libc-dev Version 2.6.32-29 is for the "amd64" architecture.
```

```
linux-source-2.6.32_2.6.32-29_all.deb
Package linux-source-2.6.32 Version 2.6.32-29 is for the "all" architecture.

README
File "README" does not appear to match the
standard .deb naming convention.

$
```

identify_debs.sh

下面是关于正则表达式测试的最后一个例子。这个脚本也能通过 RPM 包的命令规则(包名称-版本号-架构.rpm)来识别 RPM 包。注意,该脚本还能处理像 gnome-desktop 这样的带有连字符的包名称,而且不会混淆标记版本号的连字符与包名称中的连字符。



可从
WTOX.COM
下载源代码

```
$ cat identify_pkgs.sh
#!/bin/bash

for pkg in pkgs/*
do
    pkgname=`basename $pkg`
    echo $pkgname
    if [[ $pkgname =~ (.+)_(.*)_(.*)\.deb ]]; then
        echo "Package ${BASH_REMATCH[1]} Version ${BASH_REMATCH[2]} is a"
        echo " Debian package for the ${BASH_REMATCH[3]} architecture."
        echo
    elif [[ $pkgname =~ (.+)-(.)\.rpm ]]; then
        echo "Package ${BASH_REMATCH[1]} Version ${BASH_REMATCH[2]} is an"
        echo " RPM for the ${BASH_REMATCH[3]} architecture."
        echo
    else
        echo "File \"$pkgname\" does not appear to match the"
        echo "standard .deb or .rpm naming conventions."
    fi
done

$ ./identify_pkgs.sh
dbus-x11_1.2.24-4_amd64.deb
Package dbus-x11 Version 1.2.24-4 is a
  Debian package for the amd64 architecture.

firmware-linux-free_2.6.32-29_all.deb
Package firmware-linux-free Version 2.6.32-29 is a
  Debian package for the all architecture.

gnome-desktop-1.023.x86_64.rpm
Package gnome-desktop Version 1.023 is an
  RPM for the x86_64 architecture.

libgnomeprint-2.18.6-2.6.x86_64.rpm
Package libgnomeprint-2.18.6 Version 2.6 is an
```

```

RPM for the x86_64 architecture.

libgnomeui2-2.24.3-lmdv2010.1.i586.rpm
Package libgnomeui2-2.24.3 Version lmdv2010.1 is an
  RPM for the i586 architecture.

linux-headers-2.6.32-5-amd64_2.6.32-29_amd64.deb
Package linux-headers-2.6.32-5-amd64 Version 2.6.32-29 is a
  Debian package for the amd64 architecture.

linux-headers-2.6.32-5-common_2.6.32-29_amd64.deb
Package linux-headers-2.6.32-5-common Version 2.6.32-29 is a
  Debian package for the amd64 architecture.

linux-libc-dev_2.6.32-29_amd64.deb
Package linux-libc-dev Version 2.6.32-29 is a
  Debian package for the amd64 architecture.

linux-source-2.6.32_2.6.32-29_all.deb
Package linux-source-2.6.32 Version 2.6.32-29 is a
  Debian package for the all architecture.

README
File "README" does not appear to match the
standard .deb or .rpm naming conventions.
$

```

identify_pkgs.sh

5.4.5 数值测试

有 6 种数值比较测试可供使用。如果两数相等，`-eq` 返回真；如果不相等，`-ne` 返回真。`-lt` 与 `-gt` 分别用于测试两个数的小于与大于关系。如果要测试小于或等于关系，可以使用 `-le`；`-ge` 测试大于或等于的关系。下面的脚本是一个猜数字的游戏，从游戏历史中我们可以找到游戏的获胜方法。



可从
wrox.com
下载源代码

```

$ cat numberguess.sh
#!/bin/bash
MAX=50
guess=-1
let answer=($RANDOM % $MAX)
let answer+=1
ceiling=$MAX
floor=0
guesses=0

while [ "$guess" -ne "$answer" ]
do
    echo "The magic number is between $floor and $ceiling."
    echo -en " Make your guess:"

```

```

read guess
guesses=`expr $guesses + 1`
if [ "$guess" -lt "$answer" ]; then
    echo "$guess is too low"
    if [ "$guess" -gt "$floor" ]; then
        floor=`expr $guess + 1`
    fi
fi
if [ "$guess" -gt "$answer" ]; then
    echo "$guess is too high"
    if [ "$guess" -lt "$ceiling" ]; then
        ceiling=`expr $guess - 1`
    fi
fi
done
echo "You got it in $guesses guesses!"
$ ./numberguess.sh
The magic number is between 1 and 50.
Make your guess: 25
25 is too low
The magic number is between 26 and 50.
Make your guess: 37
37 is too low
The magic number is between 38 and 50.
Make your guess: 46
46 is too high
The magic number is between 38 and 45.
Make your guess: 43
43 is too low
The magic number is between 44 and 45.
Make your guess: 45
45 is too high
The magic number is between 44 and 44.
Make your guess: 44
You got it in 6 guesses!
$

```

numberguess.sh

脚本每次用猜错的数字来调整“下限”和“上限”，所以可以在上下限范围内进行高效的二分排序，直到找出正确答案。这虽然不是很有趣，但却是个介绍数值测试的好例子。



脚本使用 `-lt` 和 `-gt` 来排除所猜数字匹配正确答案的可能。然而在 `if ["$guess" -gt "$floor"]` 中使用 `-ge` 比 `-gt` 更好，因为如果玩家猜的数字是可能的最小数并且是错误的，则还是需要提高下限来反映这一点。

5.4.6 组合测试

可以使用 `&&` 和 `||` 操作符来将测试组合起来。它们分别实现测试的逻辑与和逻辑或。要

测试一个文件是否可读且大小不为 0，可以组合 `-r` 和 `-s` 测试。在本例中，哪个测试失败都没有关系；除非两个条件都为真，否则计算文件的 `md5sum` 就没有意义。`/etc/hosts` 通常可读且不为空，所以下面的脚本默认参数为 `/etc/hosts`。`/etc/shadow` 通常也不为空，但除非是 `root` 用户或者在 `shadow` 组中，否则它是不可读的，所以不可能计算它的 `md5sum`。



```
$ cat md5-if-possible.sh
#!/bin/bash
filename=${1:-/etc/hosts}

if [ -r "$filename" ] && [ -s "$filename" ]; then
    md5sum $filename
else
    echo "$filename can not be processed"
fi

# Show the file if possible
ls -ld $filename 2>/dev/null
$ ./md5-if-possible.sh /etc/hosts
785ae781cf4a4ded403642097f90a275 /etc/hosts
-rw-r--r-- 1 root root 458 Dec 3 00:23 /etc/hosts
$ ./md5-if-possible.sh /etc/shadow
/etc/shadow can not be processed
-rw-r----- 1 root shadow 1038 Nov 4 21:04 /etc/shadow
$
```

`md5-if-possible.sh`

对于 `[-r "$filename"] && [-s "$filename"]` 语法，如果文件不可读，则不会执行 `-s` 测试。利用这一特性可以写出方便但不是很容易理解的短路语句。下面的脚本只有在不显示错误消息的情况下才会运行成功，因为只有在测试已经通过的条件下才会调用 `echo`。可以通过首先执行那种速度最快或者最可能失败的测试来提高脚本的运行速度。



```
$ cat readable-and.sh
#!/bin/bash
filename=${1:-/etc/hosts}

[ -r $filename ] && echo "$filename is readable"
$ ./readable-and.sh /etc/hosts
/etc/hosts is readable
$ ./readable-and.sh /etc/shadow
$
```

`readable-and.sh`

|| 操作符执行的是逻辑或，所以它只关心满足某个条件而不关心具体满足哪个，这就是要利用的特性。



可从
wrox.com
下载源代码

```
$ cat mine.sh
#!/bin/bash
filename=${1:-/etc/hosts}

if [ -O "$filename" ] || [ -G "$filename" ]; then
    echo "$filename is mine (or my group's)"
else
    echo "$filename is not mine (nor my group's)"
fi
$ ./mine.sh /etc/hosts
/etc/hosts is not mine (nor my group's)
$ ./mine.sh $HOME
/home/steve is mine (or my group's)
$
```

mine.sh

如果-O(属主)或-G(属组)测试通过, 则整个测试通过。||操作符会执行所有测试直到某个测试通过。利用这一有用的副作用可以写出只有在测试失败条件下才运行的命令。



可从
wrox.com
下载源代码

```
$ cat readable-or.sh
#!/bin/bash
filename=${1:-/etc/hosts}

[ -r $filename ] || echo "$filename is not readable"
$ ./readable-or.sh $HOME
$ ./readable-or.sh /etc/shadow
/etc/shadow is not readable
$
```

readable-or.sh

这些短路语句很有用, 并且有必要了解它们, 因为很多 shell 脚本都使用这种语句。但是, 建议不要过度使用这一特性, 因为它们的可读性不如较长的 if/then/else 语法。有些人认为较短的形式会更快, 但事实并非如此。

我们注意到短路语句根本没有使用 if 语句, 而是直接调用 test(用[]形式)。可以将这种语法用于任意命令, [也并不是 if 语句的一部分。shell 可以对任意命令使用&&和||操作符, 而不仅仅是 if 和 test。下面给出一些使用&&和||操作符的其他命令的例子。

```
$ wc -l /etc/hosts || echo "wc failed to read /etc/hosts"
18 /etc/hosts
$ wc -l /etc/hosts.bak || echo "wc failed to read /etc/hosts.bak"
wc: /etc/hosts.bak: No such file or directory
wc failed to read /etc/hosts.bak
$ wc -l /etc/hosts | grep "^20 " && echo "/etc/hosts is a 20 line file"
$ wc -l /etc/hosts | grep "^20 " || echo "/etc/hosts is not a 20 line file"
/etc/hosts is not a 20 line file
$ wc -l /etc/hosts | grep "^18 " && echo "/etc/hosts is an 18 line file"
18 /etc/hosts
/etc/hosts is an 18 line file
$
```

5.5 case

`case` 提供了一种更加清晰、易于编写以及更具可读性的代替 `if/then/else` 的结构，尤其是在对多个值进行测试的情况下。在 `case` 语句中，我们列出要识别以及要操作的值，然后为每个值提供一个代码块。基本的 `case` 块如下所示：



可从
wrox.com
下载源代码

```
$ cat fruit.sh
#!/bin/bash

read -p "What is your favorite fruit?: " fruit
case $fruit in
    orange) echo "The $fruit is orange" ;;
    banana) echo "The $fruit is yellow" ;;
    pear) echo "The $fruit is green" ;;
    *) echo "I don't know what color a $fruit is" ;;
esac

$ ./fruit.sh
What is your favorite fruit?: banana
The banana is yellow

$ ./fruit.sh
What is your favorite fruit?: apple
I don't know what color a apple is

$
```

fruit.sh

该脚本显示了各种水果的颜色，并使用 `*` 处理任何其他水果。再看之前使用 `elif` 处理多种不同 `uname` 的代码，如果使用 `case` 会更加简洁与方便。注意，脚本没有智能到可以将文本 `a apple` 的语法错误修正过来的程度。



可从
wrox.com
下载源代码

```
$ cat uname-case.sh
#!/bin/bash
OS=`uname -s`

case "$OS" in
    FreeBSD) echo "This is FreeBSD" ;;
    CYGWIN_NT-5.1) echo "This is Cygwin" ;;
    SunOS) echo "This is Solaris" ;;
    Darwin) echo "This is Mac OSX" ;;
    AIX) echo "This is AIX" ;;
    Minix) echo "This is Minix" ;;
    Linux) echo "This is Linux" ;;
    *) echo "Failed to identify this OS" ;;
esac
```

uname-case.sh

尽管 `*` 像是一个特殊的指令，但它只是可能的最一般化的通配符，因为它绝对能匹配

任意字符串。这意味着可以使用更高级的模式匹配，并且确实如此：



可从
wrox.com
下载源代码

```
$ cat surname.sh
#!/bin/bash
read -p "What is your surname?: " surname

case $surname in
    [a-g]* | [A-G]*) file=1 ;;
    [h-m]* | [H-M]*) file=2 ;;
    [n-s]* | [N-S]*) file=3 ;;
    [t-z]* | [T-Z]*) file=4 ;;
    *) file=0 ;;
esac

if [ "$file" -gt "0" ]; then
    echo "$surname goes in file $file"
else
    echo "I have nowhere to put $surname"
fi
$ ./surname.sh
What is your surname?: Apple
Apple goes in file 1
$ ./surname.sh
What is your surname?: apple
apple goes in file 1
$ ./surname.sh
What is your surname?: Parker
Parker goes in file 3
$ ./surname.sh
What is your surname?: 'ougho
I have nowhere to put 'ougho
$
```

surname.sh

这让文档系统可以将顾客的详细信息存储在不同的文件中，具体由顾客姓氏的首字母决定。脚本还显示了对多个模式的匹配，如[A-G]*或[a-g]*指向文件1。如果设置了shell选项 `nocasematch`，则没有必要重复大小写，因为设置之后无论大小写都不进行区分了。默认情况下，与 Unix 下的大多数其他规则一样，模式匹配还是区分大小写的。我们也可以在测试之前将 `$surname` 强制转换为全大写或全小写，这样就可以避免使用 `nocasematch` 选项(这种形式可以在定义不明确的情况下更具灵活性)。

有关 `case` 的 `bash` 实现的一个鲜为人知的特性是，可以使用 `;;&` 或 `&` 而不是 `;;` 来终止语句。`;;` 表示不再执行其他语句，而如果使用 `;;&` 终止语句，则还要匹配接下来的所有模式。如果使用 `&` 终止语句，则认为接下来的模式已经匹配。下面的例子描述了所有输入(分为大写字母、小写字母、数字以及其他)。常规的 `case` 语句在第一次匹配之后就会终止。下面的代码在语句末尾使用 `;;&` 表示需要继续执行测试。



该特性是 `bash` shell 特有的，它不是 Bourne shell 的标准特性。所以如果需要编写可移植脚本，不要使用该特性，因为在其他 shell 中会导致语法错误消息。



可从
wrox.com
下载源代码

```
$ cat case1.sh
#!/bin/bash

read -p "Give me a word: " input
echo -en "You gave me some "
case $input in
    *[:lower:]*) echo -en "Lowercase " ;;&
    *[:upper:]*) echo -en "Uppercase " ;;&
    *[:digit:]*) echo -en "Numerical " ;;&
    *) echo "input." ;;
esac
$ ./case1.sh
Give me a word: Hello
You gave me some Lowercase Uppercase input.
$ ./case1.sh
Give me a word: hello
You gave me some Lowercase input.
$ ./case1.sh
Give me a word: HELLO
You gave me some Uppercase input.
$ ./case1.sh
Give me a word: 123
You gave me some Numerical input.
$ ./case1.sh
Give me a word: Hello 123
You gave me some Lowercase Uppercase Numerical input.
$ ./case1.sh
Give me a word: !@#
You gave me some input.
$
```

case1.sh

另一个 `bash` 特有的语句结束方式是 `;&`。它将导致执行接下来的代码块就如同已经匹配成功一样。我们可以使用它来使 `Ramsey Street` 表现得像已经匹配了 `Melbourne | Canberra | Sydney`，这样 `Ramsey Street` 被认为是 `Australia` 的一部分，就如同 `Melbourne`、`Canberra` 和 `Sydney` 一样。下面的例子还展示了一些与 `;&` 相关的用法。例如，可以独立于对国家的匹配之外对首都进行匹配。



可从
wrox.com
下载源代码

```
$ cat case2.sh
#!/bin/bash

read -p "Which city are you closest to?: " city
```

```
case $city in
    "New York"|London|Paris|Tokyo)
        # You can identify the capital cities and still fall through to
        # match the specific country below.
        echo "That is a capital city" ;;&
    Chicago|Detroit|"New York"|Washington)
        echo "You are in the USA" ;;
    London|Edinburgh|Cardiff|Dublin)
        echo "You are in the United Kingdom" ;;
    "Ramsey Street")
        # This is a famous street in an unspecified location in Australia.
        # You can still fall through and run the generic Australian code
        # by using the ;& ending.
        echo "G'Day Neighbour!" ;&
    Melbourne|Canberra|Sydney)
        echo "You are in Australia" ;;
    Paris)
        echo "You are in France" ;;
    Tokyo)
        echo "You are in Japan" ;;
    N*)
        # We have already matched "New York" and ended it with a ;;
        # so New York will not fall through to this test. Other places
        # beginning with N will fall through to here.
        echo "Your word begins with N but is not New York" ;;
    *)
        echo "I'm sorry, I don't know anything about $city" ;;
esac
$ ./case2.sh
Which city are you closest to?: London
That is a capital city
You are in the United Kingdom
$ ./case2.sh
Which city are you closest to?: Paris
That is a capital city
You are in France
$ ./case2.sh
Which city are you closest to?: New York
That is a capital city
You are in the USA
$ ./case2.sh
Which city are you closest to?: Nottingham
Your word begins with N but is not New York
$ ./case2.sh
Which city are you closest to?: Texas
I'm sorry, I don't know anything about Texas
$ ./case2.sh
Which city are you closest to?: Sydney
You are in Australia
$ ./case2.sh
```

```
Which city are you closest to?: Ramsey Street  
G'Day Neighbour!  
You are in Australia  
$
```

case2.sh

5.6 本章小结

本章介绍了各种控制条件执行的方法——从简单的 `if/then/else` 结构到可以完成各种不同任务的 `test`，再到更加灵活的匹配不同输入集合的 `case`。第 6 章介绍这些测试的其他用法，尤其是对循环的控制，而循环是使用这些测试的更加专门化的语言结构。

使用循环进行流控制

循环是编写代码的重要工具。编程与计算机的一般好处在于机器比人类能更快更高效地完成一些现实生活中的任务，所以经常会遇到的情况是，程序员花大量时间仔细地编写出几行代码，然后计算机重复运行这段代码的时间是编写时间的上百倍，甚至几千倍。循环的基本结构是一个要执行的代码块，以及如何停止循环与继续执行程序。本章将介绍 shell 中 4 种不同的循环结构：`for`、`while`、`until` 和 `select`。它们都有各自的用途和优缺点。对于某个特定任务应当使用何种循环一般都(也不总是)非常明显。

6.1 for 循环

与大多数循环不同，`for` 循环在每次循环执行之后不会对变量进行条件测试，而是从一个序列开始，遍历序列中的每一个元素直到终点。这一特点让 `for` 成为了最具确定性的循环结构，但并不意味着序列中的元素必须在脚本中显式地写出来(尽管可以且经常这么做)。`for` 循环能够遍历文件中的每个单词、变量中的内容，甚至是其他命令的输出。然而，它的最简形式是给出一个要遍历的元素集合。



可从
wrox.com
下载源代码

```
$ cat fruit.sh
#!/bin/bash
for fruit in apple orange pear
do
    echo "I really like ${fruit}s"
done
echo "Let's make a salad!"
$ ./fruit.sh
I really like apples
I really like oranges
I really like pears
Let's make a salad!
$
```

fruit.sh

上面的代码定义了一个变量 `fruit`，它被依次赋值为 `apple`、`orange` 和 `pear`。循环执行了 3 次，直到用完列表中的元素。循环第 1 次将 `fruit` 赋值为 `apple`，第 2 次赋值为 `orange`，第 3 次也是最后一次赋值为 `pear`。一旦循环结束，脚本继续正常执行 `done` 语句之后的命令。本例中，脚本最后显示 “Let's make a salad!” 来表示循环而不是脚本已经终止。

6.1.1 for 循环的使用时机

在已知要对一个元素集合执行相同操作，而不是重复执行某个操作直到满足某个条件时，使用 `for` 循环最合适。换用文件集合或者总是用相同的输入集合执行相同操作，这样的任务也比较适合使用 `for` 循环。如果需要在某些测试或者其他结果的基础上中断循环，则 `for` 循环就不那么适用了。

6.1.2 向 for 提供数据

上面的例子毫无意义，可以进行一些修改让它更有意义。首先，输入可以是变量的值，而不是将值硬编码在脚本中。



可从
wrox.com
下载源代码

```
$ cat fruit-var.sh
#!/bin/bash
fruits="apple orange pear"
for fruit in $fruits
do
    echo "I really like ${fruit}s"
done
echo "Let's make a salad!"
$ ./fruit-var.sh
I really like apples
I really like oranges
I really like pears
Let's make a salad!
$
```

fruit-var.sh

这样便可以灵活地编写更具交互性的脚本。`for` 循环可以从用户获取输入，可以交互式地读取输入或者从命令行本身读取。如第 3 章介绍的，`$*` 会扩展成命令行中传递的所有参数。



可从
wrox.com
下载源代码

```
$ cat fruit-read.sh
#!/bin/bash
echo -en "Please tell me some of your favorite fruit: "
read fruits
for fruit in $fruits
do
    echo "I really like ${fruit}s"
done
echo "Let's make a salad!"
$ ./fruit-read.sh
```

```

Please tell me some of your favorite fruit: kiwi banana grape apple
I really like kiwis
I really like bananas
I really like grapes
I really like apples
Let's make a salad!

```

fruit-read.sh



与第5章的 `a apple` 类似，这些脚本都太简单，不能将下面的 `cherry` 转换为复数形式 `cherries`。



可从
wrox.com
下载源代码

```

$ cat fruit-cmdline.sh
#!/bin/bash
for fruit in $*
do
    echo "I really like ${fruit}s"
done
$ ./fruit-cmdline.sh satsuma apricot cherry
I really like satsumas
I really like apricots
I really like cherries
Let's make a salad!
$

```

fruit-cmdline.sh

还可以用另一种方式处理命令行参数。语法中的 `in list` 部分是可选的。下面的脚本完全可以合法运行。它按照与 `fruit-cmdline.sh` 相同的方式处理 `$@` 变量。



可从
wrox.com
下载源代码

```

$ cat for.sh
#!/bin/bash
for fruit
do
    echo I really like $fruit
done
echo "Let's make a salad!"
$ ./for.sh apples oranges bananas
I really like apples
I really like oranges
I really like bananas
Let's make a salad!
$

```

for.sh

这一技术还适用于函数。下面的例子演示了函数中拥有的同样功能，也就是将 `$@` 替换为函数的参数。



可从
wrox.com
下载源代码

```
$ cat for-function.sh
#!/bin/bash

do_i_like()
{
    for fruit
    do
        echo I really like $fruit
    done
}

do_i_like apples bananas oranges
do_i_like satsumas apricots cherries

echo "Let's make a salad!"
$ ./for-function.sh
I really like apples
I really like bananas
I really like oranges
I really like satsumas
I really like apricots
I really like cherries
Let's make a salad!
$
```

for-function.sh

上面脚本中的过程仍然是非常手动化——所有这些循环处理的都是静态的 `fruit` 列表。Unix 的设计原则是，有很多工具，它们各自做一件事并把它做好。有很多较小的、简单的工具能用来向 `for` 循环输入数据。最理想的一个就是 `seq` 命令，将在第 14 章详细介绍。`seq` 命令只在基于 GNU 的系统中可用，如 Linux，所以尽管它非常有用，但要注意它在很多不同操作系统中是不可移植的。`seq` 可以用来监视网络中的计算机哪些响应 `ping` 命令而哪些不响应。



可从
wrox.com
下载源代码

```
$ cat ping.sh
#!/bin/bash

UPHOSTS=/var/log/uphosts.`date +%m%d%Y`
DOWNHOSTS=/var/log/downhosts.`date +%m%d%Y`
PREFIX=192.168.1
for OCTET in `seq 1 254`
do
    echo -en "Pinging ${PREFIX}.${OCTET}...."
    ping -c1 -w1 ${PREFIX}.${OCTET} > /dev/null 2>&1
    if [ "$?" -eq "0" ]; then
        echo " OK"
        echo "${PREFIX}.${OCTET}" >> ${UPHOSTS}
    else
        echo " Failed"
```

```

    echo "${PREFIX}.${OCTET}" >> ${DOWNHOSTS}
fi
done
$ ./ping.sh
Pinging 192.168.1.1.... OK
Pinging 192.168.1.2.... Failed
Pinging 192.168.1.3.... OK
Pinging 192.168.1.4.... OK
. . . etc
Pinging 192.168.1.252.... OK
Pinging 192.168.1.253.... OK
Pinging 192.168.1.254.... Failed
$

```

ping.sh

如脚本所示，反引号(`)得到 seq 命令的输出并作为 for 循环的输入。如果 seq 可以这样使用，那么其他任何命令都可以这样使用。这样就有了更多的可能性。下面的代码用管道将 grep 192.168.1 的输出传递给 awk，获得表示主机名的第二个字段。该代码并不能完全防止出错，但假设 hosts 文件具有良好的格式，应当能获取到 192.168.1.0/24 网段上的所有主机名。生成主机列表的整个管道命令如下：

```
$ grep "^192\.168\.1\." /etc/hosts | awk '{ print $2 }'
```

在这个特定的机器上，上面的命令产生了下列要测试的主机：

```

router plug declan atomic jackie goldie elvis

$ cat mynet.sh
#!/bin/bash

for host in `grep "^192\.168\.1\." /etc/hosts | awk '{ print $2 }'`
do
    ping -c1 -w1 $host > /dev/null 2>&1
    if [ "$?" -eq "0" ]; then
        echo "$host is up"
    else
        echo "$host is down"
    fi
done
$ ./mynet.sh
router is up
plug is down
declan is down
atomic is down
jackie is up
goldie is up
elvis is down
$

```

该语法的另一个用法是选择特定文件进行操作。实用程序 `mcelog` 输出机器从运行开始便检测到的机器检查异常。它的常规运行方式是 `mcelog >> /var/log/mcelog`。这意味着 `/var/log/mcelog` 通常会(希望)是空文件,但如果它不为空,则可能变得非常大,因为有故障的硬件很可能会产生大量的机器检查异常。因此,如果这些文件包含数据,最好将它们压缩。但如果对大小为 0 字节的文件用 `gzip` 进行压缩,会得到一个 34 字节的文件,这是由于 `gzip` 需要包含一些文件头来对 `gzip` 文件进行识别。这可能导致情况的恶化,所以 `for` 循环将对一个文件集合进行处理,并压缩那些大小不为 0 的文件(大小为 0 的文件在 `wc` 的输出中是一个 0 且后跟一个空格)。



可从
wrox.com
下载源代码

```
$ cat mcezip.sh
#!/bin/bash
```

```
for mce in `wc -l /var/reports/mcelogs/*.log | grep -vw "0 "`
do
    gzip $mce
done
$
```

mcezip.sh

第 4 章讨论的通配符也能用来向 `for` 循环输入数据。很多 Linux 发行版都有一个 `/etc/profile.d` 目录,其中包含了额外的登录脚本。`/etc/profile` 脚本在最后依次调用每个额外登录脚本。使用点命令来 `source` 文件,而不是简单地执行文件,所以任何环境变量的改变都能反映到调用 `shell` 中。

```
$ for i in /etc/profile.d/*.sh; do
>   if [ -r $i ]; then
>     . $i
>   fi
> done
```

值得强调的是, `for` 循环只对命令行处理一次。下面的循环不是无限循环,它只会对循环开始执行时 `/tmp` 中存在的文件进行操作。



可从
wrox.com
下载源代码

```
$ cat backup.sh
#!/bin/bash
```

```
for file in /tmp/*
do
    if [ -f ${file} ]
    then
        if [ -e "${file}.bak" ]
        then
            echo "Error: Skipping ${file}.bak as it already exists"
        else
            echo "Backing up $file"
            cp "${file}" "${file}.bak"
        fi
    fi
done
```

```

    fi
fi
done
$ ./backup.sh
Backing up /tmp/fizzbuzz-case.sh
Backing up /tmp/foo.bin
Backing up /tmp/todo
$ ls /tmp/*.bak
/tmp/fizzbuzz-case.sh.bak /tmp/foo.bin.bak /tmp/todo.bak
$ ./backup.sh
Error: Skipping /tmp/fizzbuzz-case.sh.bak as it already exists
Backing up /tmp/fizzbuzz-case.sh.bak
Error: Skipping /tmp/foo.bin.bak as it already exists
Backing up /tmp/foo.bin.bak
Error: Skipping /tmp/todo.bak as it already exists
Backing up /tmp/todo.bak
$ ls /tmp/*.bak
/tmp/fizzbuzz-case.sh.bak      /tmp/foo.bin.bak      /tmp/todo.bak
/tmp/fizzbuzz-case.sh.bak.bak /tmp/foo.bin.bak.bak /tmp/todo.bak.bak
$

```

backup.sh

下面是另一个有用的脚本，它利用 shell 遍历变量中数值的功能实现服务器的自动安装，尤其是安装群集节点。即使是在中心 DNS 服务失效的情况下，对于群集中的节点而言，相互识别的功能依然非常重要。因此，将其他群集节点名称与 IP 地址包含在本地/etc/hosts 文件中的情况很常见。即使 DNS 的作用意味着可以不必将这些信息存储在本地文件中，但是对于 DNS 失效的特殊情况，这样仍然很有用。



可从
wrox.com
下载源代码

```

$ cat cluster.sh
#!/bin/bash

NODES="node1 node2 node3"
echo >> /etc/hosts
echo "### Cluster peers" >> /etc/hosts
echo "# Added on `date`" >> /etc/hosts
for node in $NODES
do
    getent hosts $node >> /etc/hosts
done
echo "### End of Cluster peers" >> /etc/hosts
$

```

cluster.sh

在安装多个机器时，最好编写尽可能多的配置来确保构建的一致性，并减少所需的自定义。上面的脚本会将类似下面的文本添加到每个节点的/etc/hosts 文件中：

```

### Cluster peers
# Added on Fri Apr 23 14:56:43 PST 2010

```

```
192.168.1.20      node1 node1.example.com
192.168.1.21      node2 node2.example.com
192.168.1.22      node3 node3.example.com
### End of Cluster peers
```


这样做虽然简单，但却是值得的。如果某个应用程序很重要，需要群集化，则该群集应当也能够 DNS 失效的情况下继续工作。

最后关于标准 `bash` 的 `for` 循环要注意的是，它能处理没有输入的情况。在这种情况下根本就不会执行循环体。再看本章开头的 `fruit-read.sh` 脚本，可以看出它处理不同输入量的方式——首先输入两个单词，然后是一个，最后一个都没有。最后一次运行时，不输入任何单词直接按下回车键。

```
$ ./fruit-read.sh
Please tell me some of your favorite fruits: apple banana
I really like apples
I really like bananas
Let's make a salad!
$ ./fruit-read.sh
Please tell me some of your favorite fruits: apple
I really like apples
Let's make a salad!
$ ./fruit-read.sh
Please tell me some of your favorite fruits: (just press return here)
Let's make a salad!
$
```

6.1.3 C 风格的 for 循环

`bash` shell 还具有 C 风格的 `for` 循环。熟悉 C 编程语言的人都能认出 `for (i=1; i<=10; i++)` 这一结构。它与目前为止所见的 `shell` 风格差别迥异。`bash` 版本基本上是相同的，增加额外的括号使其看起来像 C 中的 `for` 循环。没有必要使用 `$` 符号来引用变量的值，`i++` 操作符是合法的，甚至可以使用由逗号隔开的多个语句，如下所示：

 可从 wrox.com 下载源代码

```
$ cat cfor.sh
#!/bin/bash

for ((i=1,j=100; i<=10; i++,j-=2))
do
    printf "i=%03d j=%03d\n" $i $j
done
$ ./cfor.sh
i=001 j=100
i=002 j=098
i=003 j=096
i=004 j=094
i=005 j=092
```

```

i=006 j=090
i=007 j=088
i=008 j=086
i=009 j=084
i=010 j=082
$

```

cfor.sh

该循环从定义 $i=1, j=100$ 开始，然后执行循环直到满足条件 $i \leq 10$ ，并且在每次循环时执行 $i++j-=2$ ，让 i 加 1，让 j 减 2。循环的每次迭代使用 `printf` 以三位数的形式显示 i 和 j 。

6.2 while 循环

shell 中的另一个最常用的循环是 `while` 循环。顾名思义，它在满足条件为真的情况下执行循环体中的代码。这在无法预测满足条件的时刻时非常有用。`while` 还可以用于一直执行某种操作直到满足某个条件为止。`while` 循环的结构是首先定义循环条件，然后是条件为真时执行的代码。下面的循环不断地将变量的值倍增直到它超过 100。



可从
wrox.com
下载源代码

```

$ cat while.sh
#!/bin/bash

i=1
while [ "$i" -lt "100" ]
do
    echo "i is $i"
    i=`expr $i \* 2`
done
echo "Finished because i is now $i"
$ ./while.sh
i is 1
i is 2
i is 4
i is 8
i is 16
i is 32
i is 64
Finished because i is now 128
$

```

while.sh


因为循环执行的条件是 i 小于 100，所以 i 到 64 时，循环回到开头的测试，并且测试通过。程序再次进入循环体，将 i 赋值为 128，当执行第 8 次测试时，测试失败。shell 退出循环，执行接下来的 `echo` 语句，显示 i 的当前值 128。所以，循环不会阻止变量超过 100，但在 i 超过 100 后拒绝执行循环。

6.2.1 while 循环的使用时机

在没有需要遍历的列表但在具有决定循环何时结束的可测试条件时，使用 `while` 循环最合适。数学计算、时间比较以及当前进程的外部状态都适于使用 `while` 循环。持续显示选项并读取用户输入的菜单系统也适合使用 `while` 循环。例如，当用户选择“quit”选项时退出菜单。另一方面，如果要对固定的列表进行操作，`for` 循环可能更合适。

6.2.2 while 循环的用法


本小节展示了 `while` 循环的各种用途以及用法。`while` 循环最常用的特性是测试循环本身的变化，如下面的代码所示，测试/24(C类)网络的 ping 结果。测试语句检测地址的最后一个字节是否小于 255(没有必要测试广播地址，并且 IPv4 地址中的每个字节不会超过 255)。循环体的最后一条语句对地址中的字节进行递增；否则循环会一直对 192.168.1.1 执行 ping 命令。从本质上而言，下面的脚本重现了已介绍过的 `for` 循环的行为。

 可从 wrox.com 下载源代码

```
$ cat while-ping.sh
#!/bin/bash
PREFIX=192.168.1
OCTET=1
while [ "$OCTET" -lt "255" ]; do
    echo -en "Pinging ${PREFIX}.${OCTET}..."
    ping -c1 -w1 ${PREFIX}.${OCTET} >/dev/null 2>&1
    if [ "$?" -eq "0" ]; then
        echo " OK"
    else
        echo "Failed"
    fi
    let OCTET=OCTET+1
done
```

while-ping.sh

`while` 循环的另一个常见用法是逐行读取文本文件的内容。下面的脚本逐行读取文件并显示与文件有关的有用信息。整个 `while read/do/done` 命令被 shell 当成单条命令(确实是单条命令)，所以 `$filename` 的重定向方式与 `read < $filename` 一样。

 可从 wrox.com 下载源代码

```
$ cat readfile.sh
#!/bin/bash
filename=$1

if [ ! -r "$filename" ]; then
    echo "Error: Can not read $filename"
    exit 1
fi

echo "Contents of file ${filename}:"
```

```

while read myline
do
    echo "$myline"
done < $filename
echo "End of ${filename}."
echo "Checksum: `md5sum $filename`"
$ ./readfile.sh
Error: Can not read
$ ./readfile.sh /etc/shadow
Error: Can not read /etc/shadow
$ ./readfile.sh /etc/hosts
Contents of file /etc/hosts:
127.0.0.1        localhost

# The following lines are desirable for IPv6 capable hosts
::1        localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters

192.168.1.3    router
192.168.1.5    plug
192.168.1.10   declan
192.168.1.11   atomic
192.168.1.12   jackie
192.168.1.13   goldie    smf    spo    sgp
192.168.1.227  elvis

192.168.0.210  dgoldie ksgp
End of /etc/hosts.
Checksum: 785ae781cf4a4ded403642097f90a275 /etc/hosts
$

```

readfile.sh

上面的例子相当简单，并没有实现比 `cat` 更有用的功能。`while` 可以使用一些方法实现更多的功能，其中之一是从文本行中读取多个单词。这种方法使用 `read` 工具，将每个单词匹配到一个变量。所有多余的单词都被赋值到最后一个变量，所以在下面的例子中，脚本读取 IP 地址、主机名，然后是别名。



可从
wrox.com
下载源代码

```

$ cat readhosts.sh
#!/bin/bash

while read ip name aliases
do
    echo $ip | grep "[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*" > /dev/null
    if [ "$?" -eq "0" ]; then
        # Okay, looks like an IPv4 address
        echo "$name is at $ip"
    fi
done

```

```

    if [ ! -z "$aliases" ]; then
        echo " ... $name has aliases: $aliases"
    fi
fi
done < /etc/hosts
$ cat /etc/hosts ←—————原始/etc/hosts 文件用于引用
127.0.0.1          localhost

# The following lines are desirable for IPv6 capable hosts
::1      localhost ip6-localhost ip6-loopback
fe00::0  ip6-localnet
ff00::0  ip6-mcastprefix
ff02::1  ip6-allnodes
ff02::2  ip6-allrouters

192.168.1.3      router
192.168.1.5      plug
192.168.1.10     declan
192.168.1.11     atomic
192.168.1.12     jackie
192.168.1.13     goldie smf spo sgp
192.168.1.227    elvis
192.168.0.210    dgoldie ksgp
$ ./readhosts.sh
localhost is at 127.0.0.1
router is at 192.168.1.3
plug is at 192.168.1.5
declan is at 192.168.1.10
atomic is at 192.168.1.11
jackie is at 192.168.1.12
goldie is at 192.168.1.13
... goldie has aliases: smf spo sgp
elvis is at 192.168.1.227
dgoldie is at 192.168.0.210
... dgoldie has aliases: ksgp$

```

readhosts.sh

上面的脚本对于 `hosts` 文件的处理更加智能化，而 `cat` 不可能像这样对文件进行解释。第3章提到过这种读取文件的方法，但关于读取到文件末尾后循环退出的实际原因却不太直观。当读取到一行文本后，内置命令 `read` 返回 0(成功)；反之如果遇到文件结束标志，则返回 -1(任意非零值表示失败)。如果 `read` 到达文件末尾后不返回不一样的值，那么就无法退出循环。

这一点暗示了 `while` 循环可以使用任意在不同条件下返回不同值的命令。`test`(或者别名 `[]`)与内置 `read` 是最常见的用于 `while` 循环的命令，但实际上任何命令都可以使用。如果时间是 12:15，则命令 `date | grep 12:15` 返回成功(返回码 0)，但如果时间是 12:16，则返回失败。下面的循环实际上没有做任何事情——其中使用了 `sleep` 命令表示循环体正在执行。`grep` 命令匹配要搜索的字符串，并将 `date` 命令的输出以副作用的形式显示出来。

```
$ while date | grep 12:15
> do
>   sleep 5
> done
Tue Dec 28 12:15:48 GMT 2010
Tue Dec 28 12:15:53 GMT 2010
Tue Dec 28 12:15:58 GMT 2010
$
```

与重定向来自文件的循环输入相关的功能是将循环的所有输出重定向至文件，而不是每个命令各自的输出。这可以极大地简化较复杂循环中的代码。下面的例子列出所有的分区/dev/sda[1-4]，说明了 echo 与 ls 命令的输出全部定向至 partitions.txt 文件。没有必要使用向文件添加内容的命令，单个>就可以，因为整个循环是一条命令，所以是一次写操作。



可从
wrox.com
下载源代码

```
$ cat while-tofile.sh
#!/bin/bash
i=1
while [ $i -lt 5 ]
do
    echo "`date` : Partition $i"
    ls -ld /dev/sda$i
    sleep 1.5
    let i=$i+1
done > partitions.txt
$ ./while-tofile.sh
$ cat partitions.txt
Tue Jan 4 21:54:48 GMT 2011 : Partition 1
brw-rw---- 1 root disk 8, 1 Jan 4 18:39 /dev/sda1
Tue Jan 4 21:54:49 GMT 2011 : Partition 2
brw-rw---- 1 root disk 8, 2 Jan 4 18:39 /dev/sda2
Tue Jan 4 21:54:51 GMT 2011 : Partition 3
brw-rw---- 1 root disk 8, 3 Jan 4 21:43 /dev/sda3
Tue Jan 4 21:54:53 GMT 2011 : Partition 4
brw-rw---- 1 root disk 8, 4 Jan 4 18:39 /dev/sda4
$
```

while-tofile.sh

另一个有用的命令是内置的:命令，或者是/bin/true 命令，它们总是返回 0，表示成功。使用它们可以让循环永远执行下去。考虑下面另外两个循环，它们测试远程主机是否响应 ping 命令。第一个循环与上面的 date 循环类似，运行到主机停止响应。第二个则不断地运行，检测主机何时又开始响应。



可从
wrox.com
下载源代码

```
$ cat ping1.sh
#!/bin/bash
host=${1:-declan}
while ping -c3 -w4 $host
```

```

do
    sleep 30
done
echo "$host has stopped responding to pings"
$ ./ping1.sh
PING declan (192.168.1.10) 56(84) bytes of data.
64 bytes from declan (192.168.1.10): icmp_req=1 ttl=64 time=1.50 ms
64 bytes from declan (192.168.1.10): icmp_req=2 ttl=64 time=1.73 ms
64 bytes from declan (192.168.1.10): icmp_req=3 ttl=64 time=1.77 ms

--- declan ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 1.502/1.671/1.775/0.120 ms
PING declan (192.168.1.10) 56(84) bytes of data.
64 bytes from declan (192.168.1.10): icmp_req=1 ttl=64 time=2.26 ms
64 bytes from declan (192.168.1.10): icmp_req=2 ttl=64 time=1.41 ms
64 bytes from declan (192.168.1.10): icmp_req=3 ttl=64 time=1.44 ms

--- declan ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 1.417/1.707/2.265/0.395 ms
PING declan (192.168.1.10) 56(84) bytes of data.

--- declan ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 2998ms

declan has stopped responding to pings
$

```

ping1.sh

下面的循环可能才是真正需要的。在永不停止的 **while** 循环中不断检测也许更有用，这样还能得知目标主机的上线时间。我们必须按下 **Ctrl-C**(下面用 **^C** 表示)来终止循环。



可从
wrox.com
下载源代码

```

$ cat ping2.sh
#!/bin/bash
host=${1:-declan}

while :
do
    ping -c3 -w 4 $host > /dev/null 2>&1
    if [ "$?" -eq "0" ]; then
        echo "`date`: $host is up"
    else
        echo "`date`: $host is down"
    fi
    sleep 30
done
$ ./ping2.sh
Wed Dec 29 12:10:57 GMT 2010: declan is up
Wed Dec 29 12:11:29 GMT 2010: declan is up

```

```

Wed Dec 29 12:12:03 GMT 2010: declan is down
Wed Dec 29 12:12:36 GMT 2010: declan is down
Wed Dec 29 12:13:08 GMT 2010: declan is up
^C
$

```

ping2.sh

一种更清晰的控制循环的方式是，完全在循环的外部对循环条件进行测试。原因可能与首先要执行的循环有关，或者只是因为为了明确控制循环而存在的控制文件。下面的代码使用的控制文件包含了要测试的主机名列表。如果主机名在文件中找不到，则测试停止。在脚本的运行中，测试进行了4分钟之后，我们将 declan 从/tmp/hosts-to-ping.txt 文件中删除掉。



可从
wrox.com
下载源代码

```

$ cat ping3.sh
#!/bin/bash
host=${1:-declan}

while grep -qw $host /tmp/hosts-to-ping.txt
do
    ping -c3 -w 4 $host > /dev/null 2>&1
    if [ "$?" -eq "0" ]; then
        echo "`date`: $host is up"
    else
        echo "`date`: $host is down"
    fi
    sleep 30
done

echo "Stopped testing $host as it has been removed from /tmp/hosts-to-ping.txt"
$ echo declan > /tmp/hosts-to-ping.txt
$ ./ping3.sh declan
Wed Dec 29 12:41:19 GMT 2010: declan is up
Wed Dec 29 12:41:53 GMT 2010: declan is down
Wed Dec 29 12:42:25 GMT 2010: declan is down
Wed Dec 29 12:42:57 GMT 2010: declan is down
Wed Dec 29 12:43:29 GMT 2010: declan is up
Wed Dec 29 12:44:01 GMT 2010: declan is up
Wed Dec 29 12:44:33 GMT 2010: declan is up
Wed Dec 29 12:45:05 GMT 2010: declan is up
Stopped testing declan as it has been removed from /tmp/hosts-to-ping.txt
$

```

ping3.sh

您可能已经注意到，尽管使用 `sleep 30` 命令来将输出消息隔开，但是时间戳显示的时间间隔超过30秒。这是因为 `ping` 命令本身运行要花几秒时间。`-c3` 标志让 `ping` 发送3个包，而 `-w 4` 则表示等待响应的时间不超过4秒。当目标主机响应时，3个 `ping` 包使循环时间增加了大约2秒。当主机下线时，则增加4秒，因为在超时前足足等待了4秒。

6.3 嵌套循环

一个循环可以存在于另一个循环中，它们甚至可以是不同种类的循环。尽管循环嵌套的层数没有实际限制，但是代码缩进会变得复杂起来，并且很快会难以跟踪循环的终止位置。嵌套循环很有用，因为可以利用上各种所需循环的最有用的特性。在下面的脚本中，`while` 循环最适于持续运行直到用户输入单词 `quit` 退出循环。在 `while` 循环内部，`for` 循环最适于遍历固定的元素集合(下面代码中的 `fruit`)。尽管 `$myfruit` 在循环结尾列出，但是在第一次迭代时它为空(`myfruit=""`)，所以第一次运行只列出 3 种水果。接下来运行时则将用户最喜爱的水果包含在列表的末尾。



可从
wrox.com
下载源代码

```
$ cat nest.sh
#!/bin/sh
myfruit=""

while [ "$myfruit" != "quit" ]
do
    for fruit in apples bananas pears $myfruit
    do
        echo "I like $fruit"
    done # end of the for loop
    read -p "What is your favorite fruit? " myfruit
done # end of the while loop
echo "Okay, bye!"

$ ./nest.sh
I like apples
I like bananas
I like pears
What is your favorite fruit? grapes
I like apples
I like bananas
I like pears
I like grapes
What is your favorite fruit? plums
I like apples
I like bananas
I like pears
I like plums
What is your favorite fruit? quit
Okay, bye!

$
```

nest.sh

6.4 循环的退出与继续

尽管利用上面这些特性可以写出非常灵活和有良好控制流的循环，但有时却需要在中

途退出循环。可以使用 `break` 命令实现循环的中途退出。该 `shell` 内置命令使程序从最内层的循环中退出。可以通过传递数值参数指定更高层的退出。默认参数为 1，表示当前循环。`break 2` 将从最内层循环中退出，同时退出包含它的外层循环。`break 3` 将退出第 2 层循环外的循环，依此类推。下面的脚本有 2 个 `for` 循环，外层循环从 1 数到 6，内层循环遍历 a、b、c、d、e 和 f。如第一次测试运行所示，内层循环在输入 1 时退出，继续遍历接下来的数字。当输入 2 时，两层循环都退出，并继续执行 `done` 之后的 `echo "That's all, folks"`。

```
$ cat break.sh
#!/bin/bash
for number in 1 2 3 4 5 6
do
    echo "In the number loop - $number"
    read -n1 -p "Press b to break out of this loop: " x
    if [ "$x" == "b" ]; then
        break
    fi
    echo
    for letter in a b c d e f
    do
        echo
        echo "Now in the letter loop... $number $letter"
        read -n1 -p "Press 1 to break out of this loop, 2 to break out totally: " x
        if [ "$x" == "1" ]; then
            break
        else
            if [ "$x" == "2" ]; then
                break 2
            fi
        fi
    done
    echo
done
echo
echo "That's all, folks"
$ ./break.sh
In the number loop - 1
Press b to break out of this loop: z

Now in the letter loop... 1 a
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 1 b
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 1 c
Press 1 to break out of this loop, 2 to break out totally: 1
In the number loop - 2
Press b to break out of this loop: z

Now in the letter loop... 2 a
```



```

Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 2 b
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 2 c
Press 1 to break out of this loop, 2 to break out totally: 1
In the number loop - 3
Press b to break out of this loop: z

Now in the letter loop... 3 a
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 3 b
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 3 c
Press 1 to break out of this loop, 2 to break out totally: 2
That's all, folks
$

```

第二次测试运行说明了在外层循环中输入 b(调用无参数的 `break`)与在内层循环中输入 2(调用 `break 2`)会将程序执行转移到同一点。



可从
wrox.com
下载源代码

```

$ ./break.sh
In the number loop - 1
Press b to break out of this loop: z

Now in the letter loop... 1 a
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 1 b
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 1 c
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 1 d
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 1 e
Press 1 to break out of this loop, 2 to break out totally: z
Now in the letter loop... 1 f
Press 1 to break out of this loop, 2 to break out totally: z
In the number loop - 2
Press b to break out of this loop: b
That's all, folks
$

```

break.sh

与 `break` 命令配套的是 `continue` 命令，它也是 shell 内置命令。`continue` 与 `break` 有关联，但不是退出当前循环，而是直接跳转到对循环进行控制的条件测试。与 `break` 类似，可以为 `continue` 指定数值参数表示要跳转的外层循环层数。在下面的脚本中，`continue` 的用法非常典型。当执行至 `continue` 时，当前循环中余下要执行的代码都忽略掉，并再次返回到循环的第一行。循环的末尾包含了一条 `echo` 语句，显示 “This is the end of the loop.”。当输入 r 立即重复执行循环时，脚本调用 `continue` 并跳过 `echo` 语句。否则，执行 `echo` 语

句，且循环正常继续运行。



可从
wrox.com
下载源代码

```
$ cat continue.sh
#!/bin/bash
i=1
while [ "$i" -lt "5" ]; do
    echo "i is $i"
    read -p "Press r to repeat, any other key to continue: " x
    let i=$i+1
    if [ "$x" == "r" ]; then
        echo "Skipping the end of the loop."
        continue
    fi
    echo "This is the end of the loop."
done
echo "This is the end of the script."
$ ./continue.sh
i is 1
Press r to repeat, any other key to continue: r
Skipping the end of the loop.
i is 2
Press r to repeat, any other key to continue: a
This is the end of the loop.
i is 3
Press r to repeat, any other key to continue: r
Skipping the end of the loop.
i is 4
Press r to repeat, any other key to continue: b
This is the end of the loop.
This is the end of the script.
$
```

continue.sh

下面代码中 `continue` 的用法有些微妙。因为在增加计数器 `i` 之前跳转至循环开头，所以可以重复执行增加计数器之前的这一部分循环体，而不改变任何变量——这样便有效地在主循环中创建一个额外循环。



可从
wrox.com
下载源代码

```
$ cat continue-backwards.sh
#!/bin/bash
i=1
while [ "$i" -lt "5" ]; do
    echo "i is $i"
    read -p "Press r to repeat, any other key to continue: " x
    if [ "$x" == "r" ]; then
        echo "Going again..."
        continue
    fi
    let i=$i+1
done
```

```
$ ./continue-backwards.sh
i is 1
Press r to repeat, any other key to continue: a
i is 2
Press r to repeat, any other key to continue: r
Going again...
i is 2
Press r to repeat, any other key to continue: b
i is 3
Press r to repeat, any other key to continue: r
Going again...
i is 3
Press r to repeat, any other key to continue: c
i is 4
Press r to repeat, any other key to continue: d
$
```

continue-backwards.sh

6.5 带 case 的 while 循环

第 5 章介绍的 `case` 语句的常见用法是用于 `while` 循环中。`case` 语句用来决定每次循环迭代要做的操作。循环本身通常使用上一节介绍的 `break` 语句退出。这在不断读取输入直到当前文本行满足某个条件或者输入结束的情况下非常适用。下面的脚本实现了一个非常简单的命令分析器，它接受 4 个命令：`echo`、`upper`、`lower` 和 `quit`。`echo` 对输入进行完整的回显，`upper` 将输入转换为大写，而 `lower` 将输入转换为小写。当输入 `quit` 命令时，则调用 `break` 退出循环。

```
$ cat while-case.sh
#!/bin/bash

quit=0
while read command data
do
    case $command in
        echo)
            echo "Found an echo command: $data"
            ;;
        upper)
            echo -en "Found an upper command: "
            echo $data | tr '[:lower:]' '[:upper:]'
            ;;
        lower)
            echo -en "Found a lower command: "
            echo $data | tr '[:upper:]' '[:lower:]'
            ;;
        quit)
            echo "Quitting as requested."
```

```

        quit=1
        break
        ;;
    *)
        echo "Read $command which is not valid input."
        echo "Valid commands are echo, upper, lower, or quit."
        ;;
    esac
done

if [ $quit -eq 1 ]; then
    echo "Broke out of the loop as directed."
else
    echo "Got to the end of the input without being told to quit."
fi
$ ./while-case.sh
Hello
Read Hello which is not valid input.
Valid commands are echo, upper, lower, or quit.
echo Hello
Found an echo command: Hello
lower Hello
Found a lower command: hello
upper Hello
Found an upper command: HELLO
quit
Quitting as requested.
Broke out of the loop as directed.

```

下面是第二次运行该脚本，输入的不是 `quit`，而是表示文件结束的 `Ctrl-D(^D)`，所以循环自行终止。因为该脚本需要在 `case` 语句中设置 `$quit` 标志，所以能够区分两种不同的退出条件。



```

$ ./while-case.sh
hello
Read hello which is not valid input.
Valid commands are echo, upper, lower, or quit.
^D
Got to the end of the input without being told to quit.
$

```

while-case.sh

6.6 until 循环

除了测试条件相反，`until` 循环与 `while` 循环在其他方面完全一样。使用它可以增强可读性，并且在某些情况下能让条件语句序列更容易编写。脚本 `until.sh` 中的代码清晰地说明了循环执行的时间长度：直到 `$a` 大于 12 或者 `$b` 小于 100。在每次循环迭代中，`$a` 增加 1，

而**\$b**减小10。



可从
wrox.com
下载源代码

```
$ cat until.sh
#!/bin/bash

read -p "Enter a starting value for a: " a
read -p "Enter a starting value for b: " b
until [ $a -gt 12 ] || [ $b -lt 100 ]
do
    echo "a is ${a}; b is ${b}."
    let a=$a+1
    let b=$b-10
done
$ ./until.sh
Enter a starting value for a: 5
Enter a starting value for b: 200
a is 5; b is 200.
a is 6; b is 190.
a is 7; b is 180.
a is 8; b is 170.
a is 9; b is 160.
a is 10; b is 150.
a is 11; b is 140.
a is 12; b is 130.
$ ./until.sh
Enter a starting value for a: 10
Enter a starting value for b: 500
a is 10; b is 500.
a is 11; b is 490.
a is 12; b is 480.
$ ./until.sh
Enter a starting value for a: 1
Enter a starting value for b: 120
a is 1; b is 120.
a is 2; b is 110.
a is 3; b is 100.
$
```

until.sh

如果要使用 **while** 循环，所有的条件语句都必须反过来：**-gt** 变成**-le**，**-lt** 变成**-ge**，且**||** 变成**&&**。代码的表达式也改变了。脚本 **without-until.sh** 持续地按 1 递增 **a** 并按 10 递减 **b**，直到 **a** 小于(或等于)12 且 **b** 大于(或等于)100。代码表达方式不是那么清晰，而且当问题空间难以描述时，代码的编写更加困难。



可从
wrox.com
下载源代码

```
$ cat without-until.sh
#!/bin/bash

read -p "Enter a starting value for a: " a
read -p "Enter a starting value for b: " b
```

```

while [ $a -le 12 ] && [ $b -ge 100 ]
do
    echo "a is ${a}; b is ${b}."
    let a=$a+1
    let b=$b-10
done
$ ./without-until.sh
Enter a starting value for a: 5
Enter a starting value for b: 200
a is 5; b is 200.
a is 6; b is 190.
a is 7; b is 180.
a is 8; b is 170.
a is 9; b is 160.
a is 10; b is 150.
a is 11; b is 140.
a is 12; b is 130.
$ ./without-until.sh
Enter a starting value for a: 10
Enter a starting value for b: 500
a is 10; b is 500.
a is 11; b is 490.
a is 12; b is 480.
$ ./without-until.sh
Enter a starting value for a: 1
Enter a starting value for b: 120
a is 1; b is 120.
a is 2; b is 110.
a is 3; b is 100.
$

```

without-until.sh

6.7 select 循环

非常适合编写菜单的循环是 `select`。它最初来自于 Kornshell，但 `bash` 中也有。`select` 循环的一个有趣的特点是它根本就没有条件测试。退出循环的唯一方式是使用 `break` 或 `exit`。下面代码中的 `select` 循环不停地执行循环体，显示提示符，并将变量设置为循环提供的值。`select` 还将 `$REPLY` 赋值为用户实际输入的数字。如果用户按下回车键，`select` 重新显示它接受的列表项。如果用户输入的是非法选项，则变量不会被赋值，所以很容易就能判断选项是否合法。当然，理解 `select` 的最好方式就是运行它。



可从
wrox.com
下载源代码

```

$ cat select1.sh
#!/bin/bash

select item in one two three four five
do
    if [ ! -z "$item" ]; then

```

```

        echo "You chose option number $REPLY which is \"$item\""
    else
        echo "$REPLY is not valid."
    fi
done
$ ./select1.sh
1) one
2) two
3) three
4) four
5) five
#? 1
You chose option number 1 which is "one"
#? 4
You chose option number 4 which is "four"
#? (enter)
1) one
2) two
3) three
4) four
5) five
#? two
two is not valid.
#? 6
6 is not valid.
#? ^C
$

```

select1.sh

这个简单的循环告诉 `select` 选择的菜单项(脚本中的 `one` 到 `five`)，且循环所做的全部就是每次测试 `$item` 变量是否被赋值。如果被赋值，则它肯定是列表中与用户输入数字相应的某个单词。用户的任何合法或非法的输入都赋值给预留变量 `$REPLY`。如果输入非法，则 `$item` 为空，所以很容易检测到。脚本显示的菜单系统虽然简单，但相对比较智能。

使用 `select` 循环可以写出比上面的简单例子更加优雅的代码。它可以使用 `PS3` 变量作为提示符，提供更友好的用户体验。如果 `PS3` 变量没有赋值，`select` 显示 `#?` 作为提示符。下面的第二个脚本使用了 `select` 循环中的更多可用选项，包括设置 `PS3` 提示符，以及适时地读取 `$REPLY` 和 `$movie` 变量。在适当的时候，`select` 循环像 `ls` 按纵栏输出数据一样对较长的选项进行环绕显示。该脚本同样说明了 `select` 与 `case` 能很好地结合起来使用。



可从
wrox.com
下载源代码

```

$ cat select2.sh
#!/bin/bash

echo "Please select a Star Wars movie; enter \"quit\" to quit,"
echo "or type \"help\" for help. Press ENTER to list the options."
echo

# Save the existing value of PS3
oPS3=$PS3

```

```

PS3="Choose a Star Wars movie: "
select movie in "A New Hope" \
    "The Empire Strikes Back" \
    "Return of the Jedi" \
    "The Phantom Menace" \
    "Attack of the Clones" \
    "Revenge of the Sith" \
    "The Clone Wars"
do
    if [ "$REPLY" == "quit" ]; then
        # This break must come before other things are run in this loop.
        echo "Okay, quitting. Hope you found it informative."
        break
    fi
    if [ "$REPLY" == "help" ]; then
        echo
        echo "Please select a Star Wars movie; enter \"quit\" to quit,"
        echo "or type \"help\" for help. Press ENTER to list the options."
        echo
        # If we do not continue here, the rest of the loop will be run,
        # and we will get a message "help is not a valid option.",
        # which would not be nice. continue lets us go back to the start.
        continue
    fi

    if [ ! -z "$movie" ]; then
        echo -en "You chose option number $REPLY, which is \"$movie,\" released in "
        case $REPLY in
            1) echo "1977" ;;
            2) echo "1980" ;;
            3) echo "1983" ;;
            4) echo "1999" ;;
            5) echo "2002" ;;
            6) echo "2005" ;;
            7) echo "2008" ;;
        esac
    else
        echo "$REPLY is not a valid option."
    fi
done

# Put PS3 back to what it was originally
PS3=$oPS3
$ ./select2.sh

Please select a Star Wars movie; enter "quit" to quit,
or type "help" for help. Press ENTER to list the options.

1) A New Hope                5) Attack of the Clones
2) The Empire Strikes Back   6) Revenge of the Sith
3) Return of the Jedi        7) The Clone Wars

```



```

4) The Phantom Menace
Choose a Star Wars movie: 2
You chose option number 2, which is "The Empire Strikes Back," released in 1980
Choose a Star Wars movie: 0
0 is not a valid option.
Choose a Star Wars movie: help

Please select a Star Wars movie; enter "quit" to quit,
or type "help" for help. Press ENTER to list the options.

Choose a Star Wars movie: (enter)
1) A New Hope                      5) Attack of the Clones
2) The Empire Strikes Back         6) Revenge of the Sith
3) Return of the Jedi              7) The Clone Wars
4) The Phantom Menace
Choose a Star Wars movie: 5
You chose option number 5, which is "Attack of the Clones," released in 2002
Choose a Star Wars movie: quit
Okay, quitting. Hope you found it informative.
$

```

select2.sh

我们再一次见证了简短的脚本要远比它看起来更加智能化。上面的例子还添加了一个额外操作——显示每部电影的发行年份。另一种实现方式可能是要为每部电影设置一些变量(发行日期、票房收入等),然后在代码中使用一个“显示”部分来输出这些变量的当前值。

利用 `select` 循环真正能做的事情远比上面介绍的多得多——绝对可以将任何操作与每个选项联系起来。下面关于 `select` 的最后一例展示了一些基本的主机列表查找功能,以简单的例子说明了很容易编写菜单来管理各种系统管理员任务或者其他最终用户任务。注意,有些选项提示用户输入更多数据,而菜单结构不会对从用户获取输入作出限制。



可从

wrox.com
下载源代码

```

$ cat select3.sh
#!/bin/bash

# Save the existing value of PS3
oPS3=$PS3
PS3="Please choose a task (ENTER to list options): "
select task in Quit "View hosts" "Edit hosts" "Search hosts" \
             "Nameservice Lookup" "DNS Lookup"
do
    if [ ! -z "$task" ]; then
        case $REPLY in
            1) echo "Goodbye."
               break
               ;;
            2) cat /etc/hosts
               ;;
            3) ${EDITOR:-vi} /etc/hosts
        esac
    fi
done

```

```

;;
4) read -p "Enter the search term: " search
   grep -w $search /etc/hosts || echo "\"$search\" Not Found."
;;
5) read -p "Enter the host name: " search
   getent hosts $search || echo "\"$search\" Not Found."
;;
6) read -p "Enter the host name: " search
   nslookup $search || echo "\"$search\" Not Found."
;;
esac
else
    echo "$REPLY is not a valid option."
fi
done

```

```

# Put PS3 back to what it was originally
PS3=$oPS3
$ ./select3.sh
1) Quit                3) Edit hosts                5) Nameservice Lookup
2) View hosts          4) Search hosts                6) DNS Lookup
Please choose a task (ENTER to list options): 4
Enter the search term: sgp
192.168.1.13 goldie smf spo sgp
Please choose a task (ENTER to list options): 6
Enter the host name: google.com
Server:                192.168.0.1
Address:               192.168.0.1#53

Non-authoritative answer:
Name:   google.com
Address: 173.194.37.104

Please choose a task (ENTER to list options): (enter)
1) Quit                3) Edit hosts                5) Nameservice Lookup
2) View hosts          4) Search hosts                6) DNS Lookup
Please choose a task (ENTER to list options): 5
Enter the host name: wwwgooglecom
"wwwgooglecom" Not Found.
Please choose a task (ENTER to list options): 1
Goodbye.
$

```

select3.sh

select 工具有用且灵活，用途不是特别广泛，但对于建立简单却具有一致性和良好弹性的用户界面来说则非常强大。它能应用于很多种情况。和 **for** 一样，还可以在 **select** 中丢掉 **in (x)** 语句。在这种情况下，**select** 会使用自己所在脚本或函数的 **\$*** 参数。

6.8 本章小结

`bash shell` 提供了 4 种循环：`for`、`while`、`until` 和 `select`(尽管 `while` 与 `until` 实际上几乎一样)。这些循环提供了更强大语言的几乎所有特性，并且在为任务选择正确的循环后，可以编写出表示相当复杂的流控制的代码，而且易于维护。

另外，`break` 与 `continue` 让正式的结构化的循环更加实用，因为在现实的编程活动中，使用其他编程语言处理这些情况会导致更加复杂的控制结构，而 `shell` 脚本则被要求比这些编程语言更具实用性。

关于在脚本中使用循环，最困难的事情可能是确定使用何种循环。`for` 循环最适于遍历预定义的元素列表。`while` 和 `until` 循环更适于持续执行代码直到某些测试条件的改变。`select` 循环则能快速实现简易的菜单系统。

第 7 章将深入介绍变量，尤其是一些原始 Bourne shell 中没有，但在 `bash`、`ksh` 和 `zsh` 中添加的有用特性。

第

7

章

变 量(续)

实际上, 每种编程语言中都存在变量; 如果没有变量, 程序的作用不会比简单地执行一系列命令大多少。有了变量, 程序可以存储数据用来循环迭代、获取用户输入以及根据变量的值完成不同的任务。本章深入介绍变量, 尤其是变量在 `bash shell` 中的使用; 标准的 Bourne shell 对于变量的使用相当有限, 而 `bash` 与其他 shell 进行了较大的扩展。

7.1 变量的用法

为变量赋值时, 变量名前不使用 `$` 符号: `variable=value`。引用变量时, 要使用 `$` 符号: `echo $variable`。实际上, `$variable` 语法是一种特殊情况, 但大多数时候是可以这样使用的。变量正确的引用方式是 `${variable}`, 所以这可以让 shell 区分 `${var}iable` (表示变量 `$var` 和随后的文本 `iable`) 与 `${variable}` (表示变量 `$variable`)。这在给变量添加后缀时非常有用, 如 `${kb}Kb is $bytes bytes, or approx ${mb}Mb`:

```
$ cat mb2.sh
echo -n "Enter a size in Kb: "
read kb
bytes=`expr $kb \* 1024`
mb=`expr $kb / 1024`
echo "${kb}Kb is ${bytes} bytes, or approx ${mb}Mb."
$ ./mb2.sh
Enter a size in Kb: 12345
12345Kb is 12641280 bytes, or approx 12Mb.
$
```

如果没有花括号, 则 `${kb}Kb` 会成为 `$kbKb`。因为没有定义名为 `kbKb` 的变量, 所以它的值为空字符串。



在字符串上下文中,未定义的变量解释成空字符串。如果将变量当成数值,则解释成0。

如果去掉花括号,脚本的运行结果如下所示:

```
$ cat mb1.sh
echo -n "Enter a size in Kb: "
read kb
bytes=`expr $kb \* 1024`
mb=`expr $kb / 1024`
echo "$kbKb is $bytes bytes, or approx $mbMb."
$ ./mb1.sh
Enter a size in Kb: 12345
    is 12641280 bytes, or approx .
$
```

虽然在本书中很难显示,但是单词 `is` 之前有一个空格。空变量表示空字符串。`$bytes` 正常显示,因为它两旁有空格,所以按照要求处理为变量 `$bytes`。对 `$kb` 和 `$mb` 的引用变成了对未定义变量 `$kbKb` 和 `$mbMb` 的引用。`shell` 无法得知程序员的意图。



变量的命名规则不多,必须以字母或下划线开头且只能包含字母、数字和下划线。点号、逗号、空格以及其他字符在变量名中都是非法的。另外,变量名的第一个字符必须是字母或下划线(不可以是数字)。

按照惯例,系统变量都是大写,其中的单词用下划线隔开: `BASH_EXECUTION_STRING`、`LC_ALL` 和 `LC_MESSAGES` 等。

尽管这一惯例没有技术上的原因,且非系统变量通常为小写,但有些 `shell` 脚本的作者依然遵循系统变量的原则对所有变量使用大写。

通常而言,包含常数、字符串以及文件名等内容的变量使用大写,而包含数字、用户输入或其他“数据”变量的变量则使用小写:

```
MESSAGES=/var/log/messages
LOGFILE=/tmp/output_log.$$
echo "$0 Started at `date`" > $LOGFILE
while read message_line from $MESSAGES
do
    echo $message_line | grep -i "USB" >> $LOGFILE
done
echo "$0 Finished at `date`" >> $LOGFILE
```

使用大小写没有“好”与“坏”的区别,但最好保持一致性。

7.1.1 变量的类型

对于大多数编程语言来说,字符串、整数、浮点数以及字符等类型是有区别的。对于 shell 而言,变量的存储方式没有实际的区别。

然而有些情况下需要区别对待——当期望一个数值时,非数值被当成 0。这可以通过一些简单的数学运算来验证——首先只用整数来验证预期的行为:

```
$ echo $((1 + 2))
3
$ x=1
$ y=2
$ echo $((x + y))
3
$
```

对于一些没有 $\$((...))$ 这样的用于简单数学运算的内置语法的 shell(如 Bourne shell)而言,进行数学计算的另一种方法是使用 `expr`:

```
$ x=1
$ y=2
$ expr $x + $y
3
$
```



为了获得可移植性,本书一般使用 `expr` 进行简单的数学运算,但也可以使用 $\$((...))$ 语法和 `let` 关键字。这 3 种方法在本小节中均有使用。

另外,它能正常运行是因为还只是对数字进行数学运算,所以一切都很清楚了。如果将变量内容替换成文本,则 shell 用数值 0 来代替文本。

```
$ x=hello
$ y=2
$ echo $((x + y))
2
$
```

shell 用 0 代替了 `hello`, 所以与下面的命令等价:

```
$ echo $((0 + 2))
2
$
```

同样地,使用 `let` 也有相同的效果。因为要被 shell 处理,所以 `let` 关键字知道在这种上下文中 `$x` 的值为 0 而不是字符串 `hello`。

```
$ x=hello
$ y=2
$ let z=$x+$y
$ echo $z
2
$
```

然而，如果使用 `expr` 进行计算，`shell` 并没有关于外部工具 `expr` 输入为数字而非字符串的先验信息，所以不会用 0 替换字符串。

```
$ x=hello
$ y=2
$ expr $x + $y
expr: non-numeric argument
$
```

`shell` 没有处理 `hello`，所以它依然是合法的 `shell` 语法，但对于 `expr` 来说则是非法的输入。

```
$ expr hello + 2
expr: non-numeric argument
$
```

另外值得注意的是，变量声明时使用的引号不会被保留，因为无论变量如何声明，处理方式还是一样(除非变量值中有空格，这时需要使用引号)：

```
$ x="hello"
$ y=2
$ echo $(( $x + $y ))
2
$ expr $x + $y
expr: non-numeric argument
$ x="hello world"
$ echo $x+$y
hello world+2
$ expr $x + $y
expr: syntax error
$ let z=$x+$y
$ echo $z
0
$
```

7.1.2 变量的长度

我们可以使用 `${#variable}` 结构计算变量中字符的数目。这与其他语言中的 `strlen()` 等类似函数等价。Bourne shell 没有这一特性，但大多数其他 shell 都具有这一特性。

```
$ myvar=hello
$ echo ${#myvar}
5
```

计算字符数目时必须使用花括号{}。\${#myvar} 会扩展成\$#和字符串 myvar。\$#是调用 shell 时参数的数目, myvar 只是文本, 所以与预期的不一样, 显示的是 0myvar。

```
$ echo ${#myvar}
0myvar
```

此处值得注意的是, 与 shell 的其他操作不同, 变量中空格的存在不会对结果造成影响。这还是因为花括号, 它在没有双引号的情况下为 shell 提供了足够的结构性信息。

```
$ myvar=hello
$ echo ${#myvar}
5
$ myvar="hello world"
$ echo ${#myvar}
11
$
```

下面的脚本将文本剪裁到特定长度, 使用续行符\标记任何跨行的文本行:



可从
wrox.com
下载源代码

```
$ cat trimline.sh
#!/bin/bash

function trimline()
{
    MAXLEN=$((LINELEN - 3)) # allow space for " \ " at end of line
    if [ "${#1}" -le "${LINELEN}" ]; then
        echo "$1"
    else
        echo "${1:0:${MAXLEN}} \\"
        trimline "${1:${MAXLEN}}"
    fi
}

LINELEN=${1:-80} # default to 80 columns
while read myline
do
    trimline "$myline"
done
$ cat gpl.txt | ./trimline.sh 50
Developers that use the GNU GPL protect your ri \
ghts with two steps: (1) assert copyright on th \
e software, and (2) offer you this License givi \
ng you legal permission to copy, distribute and \
/or modify it.
$
```

trimline.sh

该脚本每次读取一行文本，并将其传递给 `trimline` 函数。`trimline` 检查接收到的文本行是否需要进一步处理。如果整行长度小于 `${LINELEN}`，则只进行回显并继续读取文本行。否则，`trimline` 回显 `${MAXLEN}` 个字符，然后是一个空格和一个反斜线（\\——需要两个反斜线，就像\"是双引号字面值一样）。函数随后将剩余的文本传递给 `trimline` 的一个新的实例作进一步处理，防止出现文本行长度超过两“行”。

7.1.3 特殊字符串操作符

当变量解释成字符串时，有一些用于字符串的特殊操作符。如果字符串刚好是 123 或者 3.142，这些操作符同样适用，但是它们是基本的字符串操作而非数值操作。在 `bash shell` 中，`>` 和 `<` 可以用来比较字符串。这一特性使用 `bash shell` 中特有的 `[[...]]` 语法——这种语法在外部程序 `/usr/bin/test` 中不存在，`Bourne shell` 中也没有，但如果确定使用的是 `bash` 或 `ksh`，则可以使用这种语法。注意最后一个测试：按照字母表排序，20 在 4 之前，所以它不是一般的未知类型的比较，而只是字符串比较。



`ksh` 同样具有这一特性，且使用相同的语法。



可从
wrox.com
下载源代码

```
$ cat sort.sh
#!/bin/bash

if [[ "$1" > "$2" ]]; then
    echo "$2 $1"
else
    echo "$1 $2"
fi

$ ./sort.sh def abc
abc def
$ ./sort.sh hello world
hello world
$ ./sort.sh world hello
hello world
$ ./sort.sh 4 20
20 4
```

sort.sh

我们可以认为 `shell` 在内部将变量存储为字符串，但是某些情况下将它们当成整数。本章的余下部分将变量当成字符串，因为这样可以进行有趣的字符串操作。



这些特性都是 `bash`（与上文提到的 `ksh`）特有的。在 `Bourne shell` 系统中使用这些语法不会起作用。如果编写的脚本要在各种平台上运行，则要小心检查可用的特性有哪些。

7.1.4 按照长度剪裁变量字符串

很多语言(或者它们的相关库)都包含了一个名为 `substr()` 的(或功能等价的)函数,用来通过字符的位置对字符串进行剪切。`bash`(尽管又不是 Bourne shell)提供了这一特性作为其变量操作语法的一部分。



`bash` 文档中将变量写成 `parameter`, 将特殊操作符写成 `word`, 所以 `${myvar:-default}` 就表示成了抽象形式 `${parameter:-word}`。这种命名方式对于 shell 用户不是特别直观。就我个人的经验而论, 当使用更清晰易懂的单词 `variable` 来代替 `parameter` 时更有利于用户的理解。使用 `parameter` 是因为符合 `bash` 开发人员的角度——那些编写 `bash` 本身的源代码的人——但对于实际使用 shell 的用户则没有什么帮助。

传统的 `substr` 调用方法是 `substr(string, offset [,length])`, 如下所示(采用的是 C 语言):

```
myvar = "foobar";
substr(myvar, 3);           // foobar becomes bar
substr(myvar, 3, 2);        // foobar becomes ba
```

第一个 `substr` 从字符串的第 4 个字符开始提取出 3 个字符。第二个 `substr` 也是从第 4 个字符开始提取, 但是限制输出为 2 个字符。因为这一特性的 `bash` 实现方法没有使用函数, 所以语法有些不同, 但结果相同:

```
${variable:3}               # foobar becomes bar.
${variable:3:2}             # foobar becomes ba.
```

实际上可能只对日志文件中处于给定位置上包含某些文本的文本行感兴趣。例如, 可能希望修改 `diff` 命令输出的格式。`diff` 命令比较两个文件并显示之间的区别, 只存在于 `file2` 中的文本行用 `>` 标记, 只存在于 `file1` 中的文本行用 `<` 标记。

```
$ cat file1
ABC
def
$ cat file2
ABC
DEF
$ diff file1 file2
2c2
< def
---
> DEF
$
```

`diff` 是用于生成补丁的有用工具, 但是它的输出有些杂乱无章。如果只对添加或删除

的文本行感兴趣，而不关心 diff 的其他语法，可以参照下面的脚本：



可从
wrox.com
下载源代码

```
$ cat diff1.sh
#!/bin/bash

diff $1 $2 | while read diffline
do
    if [ "${diffline:0:2}" == "< " ]; then
        echo "Remove line: ${diffline:2}"
    fi
    if [ "${diffline:0:2}" == "> " ]; then
        echo "Add line: ${diffline:2}"
    fi
done
$ ./diff1.sh file1 file2
Remove line: def
Add line: DEF
$
```

diff1.sh

更简洁的表示方式是使用+和-表示文本行的添加和删除。diff2.sh 在 echo 语句中将文本替换成了+和-。输出要清晰明了得多。



可从
wrox.com
下载源代码

```
$ cat diff2.sh
#!/bin/bash

diff $1 $2 | while read diffline
do
    if [ "${diffline:0:2}" == "< " ]; then
        echo "-: ${diffline:2}"
    fi
    if [ "${diffline:0:2}" == "> " ]; then
        echo "+: ${diffline:2}"
    fi
done
$ ./diff2.sh file1 file2
-: def
+: DEF
```

diff2.sh



在某些 GNU/Linux 发行版中，/bin/sh 是 dash(不是 bash)的符号链接——Ubuntu(从 6.10 版本的 Edgy Eft 开始)和 Debian GNU/Linux(从 5.0 版本的 Lenny 开始)。dash 是比 bash 更小、更快的 shell，但拥有的特性要少一些。尽管在 Ubuntu 作出这一修改时存在一些争议，但它真正的意图是如果脚本需要 bash 特性，则需要以#!/bin/bash 而不是#!/bin/sh 开头，因为#!/bin/sh 意味着只遵循 POSIX 协议。dash 不支持子字符串。

7.1.5 从字符串末尾剪裁

除了能从字符串开头剪裁, `bash` 还有从末尾进行剪裁的功能。注意, `:`与`-4`之间的空格。这个空格是必需的; 否则与`:-`相同, 后者将在 7.3 节进行介绍。如果变量未定义, 则提供默认值。这一奇怪的现象表明, 随着时间的推移以及一些新增的特性的引入, 语法也在发展与改变。

```
${variable:-4}          # foobar becomes obar
```

上面的代码与`${variable:4}`用法相同, 只是后者是从字符串的另一端开始。我们可以获取字符串末尾的最后 `n` 个字符。此处, `foobar -4` 得到 `obar`, 正如`${variable:4}`得到 `foob` 一样。

7.1.6 使用模式剪裁字符串

数据通常会有多余的部分——前导零、空格等。`bash` 具有从字符串开头或结尾删除模式的特性。有两种方式: “贪婪”与“非贪婪”模式匹配。贪婪模式匹配是指选择匹配表达式的可能的最长模式。非贪婪模式匹配是指选择匹配表达式的可能的最短模式。

- 语法`${variable#word}`从字符串开头使用非贪婪模式匹配。
- 语法`${variable##word}`从字符串开头使用贪婪模式匹配。
- 语法`${variable%word}`从字符串结尾使用非贪婪模式匹配。
- 语法`${variable%%word}`从字符串结尾使用贪婪模式匹配。

因为这种语法使用模式匹配而非纯文本, 所以它的灵活性很高。要去掉 `xxx-xxx-xxxx` 格式的电话号码中的第一部分, 语法`${phone#*-}`将匹配正确的模式。

1. 用模式从开头剪裁字符串

我们可以将`#*-`看成“剪裁到第一个`-`”。这是从字符串开头进行的非贪婪搜索, 它将去掉号码的区号但保留号码的后两个部分。

```
$ phone="555-456-1414"
$ echo ${phone#*-}
456-1414
$
```

要只保留最后一个部分, 语法`${phone##*-}`将剪裁掉所有字符直到最后一个`-`。我们可以将`##*-`看成“剪裁到最后一个`-`”。

```
$ echo ${phone##*-}
1414
$
```

如果要剪裁掉前两个部分而保留其他部分, 用模式`*-*-`进行非贪婪搜索。它将匹配 `555-456-`, 然后保留余下部分。

```
$ echo ${phone#*-*}  
1414
```

另外，`*-*`的贪婪搜索也会保留号码的最后一个部分，因为这种模式与字符串匹配的方式只有一种。

```
$ echo ${phone##*-*}  
1414
```

2. 用模式从末尾剪裁字符串

我们可以将`%-*`看成“向前剪裁到最后一个-”。它去掉最后一个`-*`，保留号码的区号与中间部分。

```
$ echo $phone  
555-456-1414  
$ echo ${phone%-*}  
555-456
```

同样的，贪婪搜索会去掉字符串中所有的`-*`模式。这与`${phone##-*}`刚好相反。

```
$ echo ${phone%%-*}  
555
```

另外，非贪婪方法可以去掉字符串的最后两个部分。下面的代码匹配`456-1414`，保留区号。

```
$ echo ${phone%-*-*}  
555  
$
```

刚开始看上去这好像是个比较容易混淆的特性并且作用有限，另外就算已知 `bash` 可用，一般的 `shell` 脚本程序员也很少使用这一特性(有时会看到一些 `bash` 脚本明确地避免使用 `bash` 特性而使用自己构造的方法。虽然有时这表示脚本编写者不知道某一特性，但也可能是出于将脚本移植到没有 `bash` 可用的系统中的目的)。

然而，这一特性确实很有用。`Perl` 在正则表达式方面非常强大。尽管 `Perl` 在这一特殊领域总有其优势，但这一特性使 `shell` 更加接近它。如果真正需要的是通过复杂的正则表达式匹配文本，那么请考虑使用 `Perl`。否则，`bash` 的这一特性就够用了。

这里给出一个示例，考虑一个要安装不同版本的 `Veritas Volume Manager` 中的某一个版本的脚本。对于不同的版本有不同的安装要求。本例要求区分 `5.0` 版本以及其他一些版本。`5.0` 版本集合有 `5.0`、`5.0MP3`、`5.0MP3RP3` 以及 `5.0MP4`。要对它们进行匹配需要匹配 `5.0` 与后面可选的 `MP` 字符串以及任意其他文本。我们使用非贪婪模式匹配简单地将 `5.0` 之后的任意 `MP*` 字符串删除。如果剩下的是 `5.0`，则原始字符串以 `5.0` 开头。

```

if [ "${vxvm_version%MP*}" == "5.0" ]; then
; # do the necessary to install VxVM 5.0
else
; # do the necessary to install another version
fi

```

再看一个更详细的例子，考虑从各种源收集到的一个 URL 列表。尽管关于 URL 组成的定义很严格，但表示 URL 的形式很多。有时 URL 协议以 `http://`、`https://` 和 `ftp://` 等开头，有时会指定特定的端口(`https://example.com:8443/`)。有时指明路径，而有时只有域名。下面的脚本可以从 URL 中只过滤出域名部分。



可从
wrox.com
下载源代码

```

$ cat url.sh
#!/bin/bash

getdomain()
{
    url=$1

    url_without_proto=${url#*://}
    echo "$url becomes $url_without_proto"

    domain_and_port=${url_without_proto%/*}
    echo "$url_without_proto becomes $domain_and_port"

    domain=${domain_and_port%:*}
    echo "$domain_and_port becomes $domain"

    getent hosts $domain | head -1
}

for url in $*
do
    getdomain $url
done
$

```

url.sh

通过剪裁掉 URL 开头的`*://`，变量 `url_without_proto` 匹配 URL 中去掉开头`*://`的余下部分。`*://`可以是 `http://`、`https://` 和 `ftp://`。注意，像 `amazon.comhttp://ebay.com` 这样非法的 URL 也能通过上面的操作剪裁成 `ebay.com`。如果没有指定协议，则没有进行匹配，也没有任何改变。

对于第二次匹配，变量 `domain_and_port` 从字符串末尾剪裁掉所有斜线。这可以将 `en.wikipedia.org/wiki/Formula_One` 剪裁成 `en.wikipedia.org(http://协议头已经从变量 url_without_proto 中删除)`。然而，`en.wikipedia.org:8080` 是合法的 URL，没有把`:8080`去掉。

最后一次替换要删除域名后的端口号。没有必要使用贪婪模式匹配，因为分号在域名中是非法的，所以一次匹配就足够了。



IPv6 的 URL 可以包含冒号。http://fc00:30:20 可以表示地址 fc00:30 的 20 端口或者 IP 地址 fc00:30:20 的 80 端口(http 的默认端口)。IPv6 通过在 IP 部分添加方括号以示区别, 所以 http://[fc00:30:20]:80/更清晰地表示 fc00:30:20 与 80 端口。

该脚本使用 3 行高效的代码能处理所有看起来很复杂的 URL。即使是风格不同的 URL 也不在话下, 并返回每个站点的 IP 地址表示找到合法的域名。首先将 http://flickr.com/ 的前导协议与尾部的斜线去掉。同样也能处理 http://del.icio.us 这种域名之后就没有数据的 URL。

```
$ ./url.sh http://flickr.com/
http://flickr.com/ becomes flickr.com/
flickr.com/ becomes flickr.com
flickr.com becomes flickr.com
68.142.214.24 flickr.com
$ ./url.sh http://del.icio.us
http://del.icio.us becomes del.icio.us
del.icio.us becomes del.icio.us
del.icio.us becomes del.icio.us
76.13.6.175 del.icio.us
$
```

对于一些较费劲的搜索, Google 的 URL 可能会相当复杂。简单的搜索很容易; 本例中没有太多奇怪的字符。

```
$ ./url.sh www.google.com/search?q=shell+scripting
www.google.com/search?q=shell+scripting becomes www.google.com
/search?q=shell+scripting
www.google.com/search?q=shell+scripting becomes www.google.com
www.google.com becomes www.google.com
74.125.230.115 www.l.google.com www.google.com
$
```

Google 邮箱的 URL 更复杂, 但脚本还是不用进行修改。一条简单的规则便能应付一些很难预料的输入:

```
$ ./url.sh https://mail.google.com/a/steve-parker.org/#inbox/12e01805b72f4c9e
https://mail.google.com/a/steve-parker.org/#inbox/12e01805b72f4c9e
becomes mail.google.com/a/steve-parker.org/#inbox/12e01805b72f4c9e
mail.google.com/a/steve-parker.org/#inbox/12e01805b72f4c9e becomes mail
.google.com
mail.google.com becomes mail.google.com
209.85.229.19 googlemail.l.google.com mail.google.com
$
```

对更复杂的 URL 进行操作需要将参数用引号括起来, 否则 shell 会在获取 URL 之前将 & 解释成后台命令。然而, 一旦 URL 成功传递给脚本, 那些奇怪的字符, 包括 ADME:B:ONA:GB:1123 中的分号, 都不会引起任何混淆。

```
$ ./url.sh "http://cgi.ebay.co.uk/ws/eBayISAPI.dll?ViewItem&item=
270242319206&ssPageName=ADME:B:ONA:GB:1123"
http://cgi.ebay.co.uk/ws/eBayISAPI.dll?ViewItem&item=270242319206&
ssPageName=ADME:B:ONA:GB:1123 becomes cgi.ebay.co.uk/ws/eBayISAPI
.dll?ViewItem&item=270242319206&ssPageName=ADME:B:ONA:GB:1123
cgi.ebay.co.uk/ws/eBayISAPI.dll?ViewItem&item=270242319206&ssPageName=
ADME:B:ONA:GB:1123 becomes cgi.ebay.co.uk
cgi.ebay.co.uk becomes cgi.ebay.co.uk
66.135.202.12 cgi-intl.ebay.com cgi.ebay.co.uk
$
```

作为最后一个测试, 让我们来看看带有端口和尾部数据的 https 的 URL。我们的简易脚本 url.sh 可以进行似乎只有 Perl 才能做的字符串分析。

```
$ ./url.sh https://127.0.0.1:6789/login.jsp
https://127.0.0.1:6789/login.jsp becomes 127.0.0.1:6789/login.jsp
127.0.0.1:6789/login.jsp becomes 127.0.0.1:6789
127.0.0.1:6789 becomes 127.0.0.1
127.0.0.1 localhost
$
```

对于脚本中的 3 次替换操作, 如果没有匹配, 则字符串不会有变化。这倒无关大碍。脚本正确处理了很多种场景, 而且确实是只用了 3 行代码。

7.2 字符串查找

sed(流编辑器)提供了一种灵活的用来替换文本的查找替换功能。例如, 可以通过将 Wintel 替换成 Linux 来一次性升级数据中心:

```
sed s/Wintel/Linux/g datacenter
```

sed 极其强大, 但 bash 也具有基本的查找替换功能。生成额外的进程比较花费时间, 尤其是在循环次数为 10、100 或 1000 的循环中。因为 sed 对于简单的文本替换而言是个相对较大的程序, 所以使用内置的 bash 功能则效率更高。

bash 的内置查找替换语法与 sed 差别不大。对于 \$datacenter 是变量, 且任务依然是将 Wintel 替换成 Linux 的情况, 上面的那一行 sed 代码与下面的代码等价:

```
echo ${datacenter/Wintel/Linux}
```

7.2.1 查找与替换

考虑/etc/passwd 中用户 Fred 的那一行。我们可以使用查找替换语法将模式 fred 修改为

wilma。

```
$ user=`grep fred /etc/passwd`
$ echo $user
fred:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$ echo ${user/fred/wilma}
wilma:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$
```

这只会修改第一个 fred。如果要将所有的 fred 修改为 wilma，应将第一个斜线/改为双斜线/(或者如文档中介绍的那样，在查找字符串的开头再加上一个斜线)。这样可以将所有要查找的模式实例替换成新文本。

```
$ echo ${user/fred/wilma}
wilma:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$ echo ${user//fred/wilma}
wilma:x:1000:1000:Fred Flintstone:/home/wilma:/bin/bash
$
```

上面的代码将用户名与主目录字段从 fred 改为 wilma，但是没有修改 GECOS 字段(这是人类可读的部分，依然是 Fred Flintstone)。bash 的这一特性不能直接对它进行修改。

如果模式必须在变量值的开头进行匹配，则要使用标准的正则表达式/^fred/wilma。然而，该 shell 没有使用正则表达式，而且脱字符号(^)用于修改变量的大小写。正确的语法是使用#而非^。本例中，最后试图将 1000 替换成 1001，但是没有匹配，因为 1000 不在一行的开头。

```
$ echo ${user/^fred/wilma}
fred:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$ echo ${user/#fred/wilma}
wilma:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$ echo ${user/1000/1001}
fred:x:1001:1000:Fred Flintstone:/home/fred:/bin/bash
$ echo ${user/#1000/1001}
fred:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$
```

类似地，要在一行的末尾进行匹配，则应在查找字符串的开头使用%，而不是像正则表达式那样在查找字符串的末尾使用\$。下面的代码将 bash 修改为 ksh：

```
$ echo ${user/%bash/ksh}
fred:x:1000:1000:Fred Flintstone:/home/fred:/bin/ksh
$
```

这一特性还可以用于更实际的用途，如修改文件扩展名。扩展名可以包含任何内容(可以在扩展名中重复任意多次)，包括要查找的模式。如果要将所有的*.TXT 文件重命名为 Unix 风格的*.txt，应使用/%.TXT/.txt：

```
#!/bin/bash
for myfile in *.TXT
do
    mynewfile=${myfile/%.TXT/.txt}
    echo "Renaming $myfile to ${mynewfile} ..."
    mv $myfile $mynewfile
done
```

因为文件列表可能包含 FILE.TXT.TXT，使用%符号可以确保只替换最后的.TXT。FILE.TXT.TXT 则被重命名为 FILE.TXT.txt，而不是 FILE.txt.txt。保留句点可以确保 fileTXT 不会被修改为 filetxt，但是 file.TXT 会被修改为 file.txt。

7.2.2 模式替换

查找模式也可以包含通配符，但只能是“贪婪匹配”的形式。将 f*d 替换成 wilma 将尽可能地匹配星号。在本例中，星号将匹配 red:x:1000:1000:Fred Flintstone:/home/fre。然而，fred 可以用 f??d 匹配。

```
$ echo $user
fred:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$ echo ${user/f*d/wilma}
wilma:/bin/bash
$ echo ${user/f??d/wilma}
wilma:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$
```

7.2.3 模式删除

如果只要删除模式，则不提供替换文本即可(而且最后的斜线/也是可选的)。相同的模式匹配规则依然适用，单个斜线/只匹配一个模式实例，/#只在一行的开头进行匹配，/%只在一行的末尾进行匹配，//匹配模式的所有实例。

```
$ echo ${user/fred}
:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$ echo ${user/#fred}
:x:1000:1000:Fred Flintstone:/home/fred:/bin/bash
$ echo ${user//fred}
:x:1000:1000:Fred Flintstone:/home/:/bin/bash
$ echo ${user/%bash}
fred:x:1000:1000:Fred Flintstone:/home/fred:/bin/
$
```

7.2.4 大小写转换

bash shell 还可以进行大小写之间的转换。要转换为大写，使用\${variable^^}；要转换为小写，使用\${variable,,}。

```
$ echo ${user^^}
FRED:X:1000:1000:FRED FLINTSTONE:/HOME/FRED:/BIN/BASH
$ echo ${user,,}
fred:x:1000:1000:fred flintstone:/home/fred:/bin/bash
$
```

另外还有一种语法，用`${variable^pattern}`将模式转换为大写，用`${variable,,pattern}`将模式转换为小写。然而，这种语法只能对一个单词进行操作，而不能对整个变量进行。

7.3 提供默认值

我们通常都希望变量包含值，但是并不知道它当前是否被赋值。如果要考虑的变量没有被赋值，为它提供一个默认值则很有用。

如果我们要编写脚本让用户编辑一个文件，可以使用下面的代码：

```
echo "You must now edit the file $myfile"
sleep 5
vi "${myfile}"
echo "Thank you for editing $myfile"
```

但不是每个人都喜欢使用 `vi`。他们可能希望使用 `emacs`、`nano`、`gedit`、`kate` 或者其他编辑器。

如果要使用其他编辑器，可以让用户设置变量(通常为 `EDITOR`)来定义他们喜欢的编辑器。将下面的代码

```
vi "${myfile}"
```

改成

```
${EDITOR} "${myfile}"
```

这便很好地解决了这一问题。用户可以定义他/她喜欢的编辑器，并且脚本更加灵活。但是如果 `EDITOR` 没有被赋值会怎样？因为变量无须声明，所以不会有错误发生。`$EDITOR` 只是被悄然替换为空字符串：

```
"${myfile}"
```

如果`$myfile`是`/etc/hosts`，则系统将尝试运行`/etc/hosts`。这会失败，因为`/etc/hosts`没有设置(或者至少不应当设置)可执行位。

```
$ ls -l /etc/hosts
-rw-r--r-- 1 root root 1494 2008-05-22 00:44 /etc/hosts
```

如果`$myfile`是`rm -rf /`，则它会被执行，而不是被编辑，与一开始的意图大相径庭。

可以为变量提供一个默认值来避免这种情况，其语法为 `${parameter:-word}`。花括号是必需的，*word* 可以在需要的时候由 shell 进行扩展。因此，如果没有设置其他编辑器，下面的代码将提供默认的 `/usr/bin/vim`：

```
${EDITOR:-/usr/bin/vim} "${myfile}"
```

如果不确定 `vim` 的路径是否总为 `/usr/bin/vim`，可以使用 `which` 命令来找到 `vim` 的位置。shell 会在反引号中将命令进行扩展，所以下面的代码更加健壮：

```
${EDITOR:-`which vim`} "${myfile}"
```


每次都这么做会显得比较麻烦，所以 shell 允许在必要的时候通过 `${parameter:=word}` 为变量设置默认值。这样一来，往后代码对变量的引用就无须进行特殊处理了：

```
${EDITOR:=`which vim`} "${myfile}"
${EDITOR} "${yourfile}"
```

最好将脚本调用的参数在脚本中替换成名称含有意义的参数，因为这样有利于脚本的可读性。例如，下面的脚本检查文件系统的使用情况，默认为根(/)文件系统。

```
#!/bin/sh
FILESYS=${1:-/}
df -h ${FILESYS}
```

从更实用的角度看，我们可能希望脚本用自定义名称的文件名来存储日志，但如果不提供文件名，则为日志文件提供一个默认的唯一(或者说可以认为是唯一的)名称。通常使用当前时间作为这个唯一的文件名，但是脚本不能每次写日志文件时都重新计算日期和时间——这样会导致每一分钟都写到一个不同的文件。默认值对于这种情况非常合适。在下面的脚本中，`LOGFILE` 被默认赋值为当前日期和时间，或者赋值为命令行中给定的路径。

 可从 wrox.com 下载源代码

```
$ cat default.sh
#!/bin/bash
LOGFILE=${1:-/tmp/log.`basename $0`-`date +%h%d.%H%M`}
echo "Logging to $LOGFILE"
$ ./default.sh
Logging to /tmp/log.default.sh-Feb13.2313
$ ./default.sh /tmp/logfile.txt
Logging to /tmp/logfile.txt
$
```

default.sh

3.1.5 节介绍了变量的两种删除方法。可以使用 `-z` 测试检测变量的状态来查看其长度是否为 0：

```
$ myvar=hello
$ if [ -z "${myvar}" ]
> then
>   echo "myvar is empty"
> else
>   echo "myvar is set to $myvar"
> fi
myvar is set to hello
```

上面的测试结果与预期一致。将变量赋值为空，然后再进行测试。

```
$ myvar=
$ if [ -z "${myvar}" ]
> then
>   echo "myvar is empty"
> else
>   echo "myvar is set to $myvar"
> fi
myvar is empty
```

比较一下将变量删除的情况。此时它不具有任何长度，因为它在环境中不存在。

```
$ unset myvar
$ if [ -z "${myvar}" ]
> then
>   echo "myvar is empty"
> else
>   echo "myvar is set to $myvar"
> fi
myvar is empty
```

-z 测试无法区别空变量与没有被赋值的变量。对于这种特殊情况，对变量使用?操作符很方便。在变量没有被赋值的情况下，可以用它来显示其他文本(如本例中的 `goodbye`)。注意，不会只显示文本，还有没有被赋值的变量的引用。

```
$ myvar=hello
$ echo ${myvar?goodbye}
hello
$ myvar=
$ echo ${myvar?goodbye}

$ unset myvar
$ echo ${myvar?goodbye}
bash: myvar: goodbye
$
```

类似的情况可以用+操作符处理。在下面的代码中，如果\$myvar 有值，则加号后的表达式将被求值；否则返回空字符串。加号后的表达式可能是 `goodbye` 这样的字符串或者是

\$x 这样的表达式。

```
$ myvar=hello
$ echo ${myvar+goodbye}
goodbye
$ unset myvar
$ echo ${myvar+goodbye}

$ x=1
$ echo ${myvar+$x}

$ myvar=hello
$ echo ${myvar+$x}
1
$
```

这一特性只能用于在某些条件下替换给定值。例如，如果一个安装程序有可选的图形化模式，且只有在变量 DISPLAY 被赋值的情况下才启用该模式，那么下面对它的调用只有在可能成功时才会启用图形化模式。下面的演示用 echo 来显示启动的命令。

```
$ unset DISPLAY
$ echo ./installer ${DISPLAY+"--display $DISPLAY"}
./installer
$ DISPLAY=127.0.0.1:0
$ echo ./installer ${DISPLAY+"--display $DISPLAY"}
./installer --display 127.0.0.1:0
$
```

7.4 间接操作

bash shell 中一个特别有用的技巧是间接操作。使用它时必须小心，因为容易将变量混淆，但它确实作用不小。

我们可以用一个变量名访问另一个变量的值，如下面的代码所示：

```
for mything in PATH GDMSESSION HOSTNAME
do
    echo $myvar is ${!myvar}
done
```

运行结果如下：

```
PATH is /usr/bin:/bin:/usr/local/bin:/home/steve/bin:/usr/games
GDMSESSION is gnome
HOSTNAME is Declan
```

起初看来似乎用处不大。但这意味着可以随时创建自己的变量名，然后用它访问数据：



可从
wrox.com
下载源代码

```
$ cat empdata.sh
#!/bin/bash
# Employee Data
Dave_Fullname="Dave Smith"
Dave_Country="USA"
Dave_Email=dave@example.com

Jim_Fullname="Jim Jones"
Jim_Country="Germany"
Jim_Email=jim.j@example.com

Bob_Fullname="Bob Anderson"
Bob_Country="Australia"
Bob_Email=banderson@example.com

echo "Select an Employee:"
select Employee in Dave Jim Bob
do
    echo "What do you want to know about ${Employee}?"
    select Data in Fullname Country Email
    do
        echo $Employee                # Jim
        echo $Data                    # Email
        empdata=${Employee}_${Data} # Jim_Email
        echo "${Employee}'s ${Data} is ${!empdata}" # jim.j@example.com
        break
    done
done
break
done
```

empdata.sh

```
$ ./empdata.sh
Select an Employee:
1) Dave
2) Jim
3) Bob
#? 2
What do you want to know about Jim?
1) Fullname
2) Country
3) Email
#? 3
Jim
Email
Jim's Email is jim.j@example.com
$
```

该脚本使用 `select` 循环提供了一个基本菜单，目的只是以简单的方式从用户获取数据。比较巧妙的是通过另外两个变量的值来定义 `$empdata` 变量的那一行代码，以及接下来使用

\$empdata 间接访问变量值的那一行代码。

7.5 使用 source 命令加载变量

存储数据尤其是配置选项的有用方法是将这些变量定义逐条记录到文本文件中。这很容易编辑，也便于 shell 脚本读取。用户一般不会意识到文件是如何被使用的。例如，基于 Red Hat 的 Linux 发行版中的/etc/sysconfig 目录。其中有个常见的用于基本网络配置的/etc/sysconfig/network 文件，它会被很多系统脚本读取。还有一些用于特定程序的文件，如/etc/sysconfig/ntpd，它包含网络时间协议(Network Time Protocol, NTP)守护进程的特定选项。考虑/etc/init.d/ntpd 文件，它在系统启动时运行。它会 source 所有这些文件，然后使用这些文件中的变量。

```
# Source function library.
. /etc/init.d/functions

# Source networking configuration.
. /etc/sysconfig/network

if [ -f /etc/sysconfig/ntpd ];then
    . /etc/sysconfig/ntpd
fi
```

配置文件本身与下面的示例类似。这些文件很容易被系统管理员或软件修改，并且在系统启动需要这些文件时，其中的变量包含了所有相关选项。

```
root@redhat# cat /etc/sysconfig/network
NETWORKING_IPV6=no
HOSTNAME=redhat.example.com
NETWORKING=yes
GATEWAY=192.168.32.1
root@redhat# cat /etc/sysconfig/ntpd
# Drop root to id 'ntp:ntp' by default.
OPTIONS="-u ntp:ntp -p /var/run/ntpd.pid"

# Set to 'yes' to sync hw clock after successful ntpdate
SYNC_HWCLOCK=no

# Additional options for ntpdate
NTPDATE_OPTIONS=""
root@redhat#
```

/etc/init.d/ntpd 脚本随后使用\$OPTIONS 选项调用二进制文件 ntpd。\$OPTIONS 在此处是值为-u ntp:ntp -p /var/run/ntpd.pid 的变量。系统管理员可以编辑文本文件/etc/init.d/ntpd，而无须编辑初始化脚本本身。这对于系统管理员来说更方便，而且因为初始化脚本本身不用修改，所以在发行版将初始化脚本作为系统升级的一部分一起升级时，相关的配置不会

丢失。

7.6 本章小结

变量是编程语言的基本组成部分。虽然 shell 语法有时会比较复杂，但它提供了在其他语言中作为算术运算或字符串操作的强大功能。shell 的变量语法可以获取子字符串、使用基本的正则表达式，以及直接对变量本身进行操作。这意味着不需要类似 `substr()`、`strlen()` 这样的很多函数。

第 3 章几乎介绍了所有的预定义变量，如 `PATH`、`PS1`、`COLUMNS`、`ROWS`、`RANDOM`、`$0`、`$1`、`$2` 等，还有 `$*` 和 `$@`，以及它们的用途。本章进一步介绍用户定义变量的用法，以及 shell 本身可以对变量值进行的强大的转换功能。

凡是有点作用的脚本都会使用变量，所以能很好地理解变量的基础知识非常重要。对于更复杂的语法而言，手头最好有一个基本的参考手册。

第 8 章介绍函数和库。函数和库将会大量使用到变量。函数与变量的组合可以实现真正意义上的计算活动。

函数和库

函数在编程语言中非常有用。定义为函数的特定代码块可以在脚本的不同位置进行重用，使同一功能具有一致性、可读性与可维护性。通常将函数的集合捆绑为库，使其在合适的位置提供相关的功能。本章介绍了函数的各种用法、shell 函数的定义，然后介绍了函数库。

8.1 函数

函数将脚本中的特殊代码块分割成较小的、更具可读性的代码块，使代码模块化。这是一种较好的设计原则，因为函数有如下特点：

- 在主脚本中隐藏实现细节，简化 shell 脚本的主体代码。
- 允许在脚本中对代码进行一致性重用，甚至在多个 shell 脚本之间重用。
- 如果工作细节发生改变，可以进行函数替换。
- 可以作为较大脚本中的较小代码块进行重复多次的不同输入值的测试，用以证明代码的正确性。

所有这些特点可以让代码更加灵活、可读与可维护。



在开发函数代码时，最好将其单独放到一个文件中，然后让脚本的“主体”用不同测试值重复调用该函数。这可以极大地加快开发与测试周期。

8.1.1 函数定义

按照惯例，函数定义在文件开头，不过这并不是硬性要求。对函数定义的结尾以及实际代码运行起点进行标记可有助于函数的调试与维护，还能让程序员在查看主代码之前不必阅读所有的函数定义。定义为函数的代码块可以用 3 种不同的方式进行声明，这取决于实际正在使用的 shell。标准的 Bourne shell 语法在函数名后面紧跟一对圆括号()与包含代码

本身的花括号{ }。



可从
wrox.com
下载源代码

```
$ cat myfunction.sh
#!/bin/bash

myfunction()
{
    echo "This is the myfunction function."
}

# Main code starts here

echo "Calling myfunction..."
myfunction
echo "Done."
$ ./myfunction.sh
Calling myfunction...
This is the myfunction function.
Done.
$
```

myfunction.sh

还有第二种语法，该语法尽管在 `bash` 与 `ksh` 下可以使用，但它不被 `Bourne shell` 接受。这种语法在函数名之后没有圆括号，而是在函数名之前使用关键字 `function`：

```
function myfunction
```

此外，还有一种只有 `bash` 接受的语法，将关键字 `function` 与圆括号组合起来。

```
function myfunction()
```

因为函数的 `Bourne shell` 语法适用于所有 `shell`，所以它最常用。第二种语法也经常使用，而且由于使用了 `function` 关键字，使得函数声明更加清晰。

8.1.2 函数输出

一些编程语言区分过程与函数。函数有返回值，并且不应当有其他副作用，而过程没有返回值，但可能有副作用。与 `C` 语言一样，`shell` 不进行这样的区分——没有过程这个概念，尽管函数可以没有返回值(这种情况下一般返回 0)，或者调用者不去检查函数的返回值。然而实际上，`shell` 函数被更多地用成过程而不是函数。部分原因在于 `shell` 函数只能返回单个字节，它是由 `$?` 变量来表示的 0~255 之间的整数，这便限制了函数的数学功能。另外，与程序和 `shell` 脚本一样，返回码经常用来表示函数中的任务是否运行成功。

1. 返回码

从函数获取值的最简单与最常见的方法是将一个简单的数字——有时表示为负数——作为表示成功(0)与失败(非 0)的返回值。下面的函数根据可能遇到的错误条件返回相关的诊断信息。这个脚本试图找到与 `eth0`、`eth1` 和 `eth2` 相关的 IP 地址。其中函数的运行方式

是针对 Red Hat Linux 类型的网络接口定义的。这种类型的网络接口定义存储在 `/etc/sysconfig/network-scripts/ifcfg-eth*` 中。变量 `IPADDR` 定义在这些文件中，所以如果有这些变量，则函数将找到并报告它们的存在。



可从

wrox.com
下载源代码

```

debian$ cat redhat-nics.sh
#!/bin/sh

getipaddr()
{
    cd /etc/sysconfig/network-scripts || return 1
    if [ -f ifcfg-$1 ]; then
        unset IPADDR
        . ifcfg-$1
        if [ -z "$IPADDR" ]; then
            return 2
        else
            echo $IPADDR
        fi
    else
        return 3
    fi
    # Not strictly needed
    return 0
}

for thisnic in eth0 eth1 eth2
do
    thisip=`getipaddr $thisnic`
    case $? in
        0) echo "The IP Address configured for $thisnic is $thisip" ;;
        1) echo "This does not seem to be a RedHat system" ;;
        2) echo "No IP Address defined for $thisnic" ;;
        3) echo "No configuration file found for $thisnic" ;;
    esac
done

```

redhat-nics.sh

因为不同的 Linux 发行版使用不同的机制来对网络进行配置，所以该脚本在不是使用 Red Hat 类型网络配置的系统中不适用。较好的方法是查看 `/etc/redhat-release` 是否存在。这个文件应当在任何 Red Hat 或其衍生系统中存在，如 CentOS 或 Oracle Linux。

```

debian$ ./redhat-nics.sh
cd: 29: can't cd to /etc/sysconfig/network-scripts
This does not seem to be a RedHat system
cd: 29: can't cd to /etc/sysconfig/network-scripts
This does not seem to be a RedHat system
cd: 29: can't cd to /etc/sysconfig/network-scripts
This does not seem to be a RedHat system
debian$

```

与预期的一致, 该函数在 Debian 系统中每次运行都失败。Debian 将网络配置存储在 `/etc/network/interfaces` 中, 与 Red Hat 完全不同。当在基于 Red Hat 的系统中运行时, 函数会报告 `eth0` 和 `eth1` 的值, 而对于 `eth2`, 尽管它存在但没有定义好的 IP 地址。

```
rhel6$ ./redhat-nics.sh
The IP Address configured for eth0 is 192.168.3.19
The IP Address configured for eth1 is 10.201.24.19
No IP Address defined for eth2
rhel6$
```



因为该循环的运行次数可能不止一次, 所以每次运行时删除 `$IPADDR` 都很重要。否则, `[-z "$IPADDR"]` 测试会通过 `eth2` 测试, 但是它包含的仍然是 `eth1` 的 IP 地址。

2. 返回字符串

您可能已经注意到, 之前的例子在查找到匹配之后通过回显 `$IPADDR` 的值来进行响应。回显内容被调用脚本获取, 因为函数调用方式是 `thisip=`getipaddr $thisnic``。这条语句的实际含义是变量 `thisip` 被赋值为函数的任何所有输出。该函数除了在找到 IP 地址后进行输出以外没有任何响应, 因此它可以提供到调用者的两个通信信道。返回码告诉调用者是否找到 IP 地址(以及没有找到情况下失败的原因), 但是输出本身才是调用者需要的实际数据。

8.1.3 写入文件

函数可以将输出写入到文件中。下面脚本中的函数简单地将第二个参数(一个数字)写入到第一个参数(一个文件名)中。调用脚本首先将 1、2、3 写入到 `file1`, 将 2、3、4 写入到 `file2`。然后将 `file1` 清空, 再次调用函数, 将 11、12、13 写入到 `file1`, 将 12、13、14 写入到 `file2`。

在这个例子中, `file2` 的名称是 `mktmep` 随机生成的独一无二的文件名, `file1` 文件名总是 `/tmp/file.1`。还可以用一个进程打开这个文件, 前提是假设它知道要访问的文件名。在另一个窗口中, 运行脚本之前先运行 `tail -F /tmp/file.1`, 然后观察输出。`tail -F` 的输出在脚本输出之后显示。



可从
wrox.com
下载源代码

```
$ cat writetofile.sh
#!/bin/sh

myfunc()
{
    thefile=$1
    echo Hello number $2 >> $thefile
}

file1=/tmp/file.1
```

```
file2=`mktemp`

for i in 1 2 3
do
    myfunc $file1 $i
    myfunc $file2 `expr $i + 1`
done

echo "FILE 1 says:"
cat $file1
echo "FILE 2 says:"
cat $file2

> $file1

for i in 11 12 13
do
    myfunc $file1 $i
    myfunc $file2 `expr $i + 1`
done

echo "FILE 1 says:"
cat $file1
echo "FILE 2 says:"
cat $file2

rm -f $file1 $file2
$ ./writetofile.sh
FILE 1 says:
Hello number 1
Hello number 2
Hello number 3
FILE 2 says:
Hello number 2
Hello number 3
Hello number 4
FILE 1 says:
Hello number 11
Hello number 12
Hello number 13
FILE 2 says:
Hello number 2
Hello number 3
Hello number 4
Hello number 12
Hello number 13
Hello number 14
$
```

writetofile.sh



`tail -F` 在文件不存在时会进行报告，但命令继续执行。另外，如果 `tail` 进程在 11、12、13 写入到文件之后才开始，则它不会获取到输出的 1、2、3，因为脚本已经在之前对文件进行了剪裁。

```
$ tail -F /tmp/file.1
tail: cannot open `/tmp/file.1' for reading: No such file or directory
tail: `/tmp/file.1' has become accessible
Hello number 1
Hello number 2
Hello number 3
tail: /tmp/file.1: file truncated
Hello number 11
Hello number 12
Hello number 13
tail: `/tmp/file.1' has become inaccessible: No such file or directory
```

另外，一些更复杂的编程语言允许从函数返回数值的复杂集合。`shell` 没有这个功能，但是可以通过输出到文件来进行类似的模拟。下面这个简单的函数计算给定数的平方与立方，然后将结果写入到一个临时文件。因为有两行写入到文件中，所以 `head -1` 与 `tail -1` 足以将这两项数据从函数中提取出来。还可以读取到两个变量中来实现相同的效果，就如同这两个变量直接被函数赋值一样。这样做虽然不是很优美，但可以达到所需的目的。



可从
wrox.com
下载源代码

```
$ cat square-cube.sh
#!/bin/bash

squarecube()
{
    echo "$2 * $2" | bc > $1
    echo "$2 * $2 * $2" | bc >> $1
}

output=`mktemp`
for i in 1 2 3 4 5
do
    squarecube $output $i
    square=`head -1 $output`
    cube=`tail -1 $output`
    echo "The square of $i is $square"
    echo "The cube of $i is $cube"
done
rm -f $output
$ ./square-cube.sh
The square of 1 is 1
The cube of 1 is 1
The square of 2 is 4
```

```

The cube of 2 is 8
The square of 3 is 9
The cube of 3 is 27
The square of 4 is 16
The cube of 4 is 64
The square of 5 is 25
The cube of 5 is 125
$

```

[squarecube.sh](#)

8.1.4 整个函数的输出重定向

不是特意将函数写入文件，而是让函数像一般情况那样写入到 `stdout`，然后将整个函数导向一个文件，这种做法可能更合适。这也意味着函数的运行可能也可能不重定向，只是取决于调用方式。下面这个简短的函数获取了 `lspci`(它显示系统中每个 PCI 设备的 PCI 路径、一个空格以及厂商的标识字符串)与 `lscpu`(它显示 CPU 的各种属性，显示方式为属性:值的格式)。替换掉 `lspci` 的第一个空格与 `lscpu` 的第一个冒号，这样生成了一个逗号分隔值(.csv)文件。这种格式的文件在电子表格中查看时可以很好地进行格式化。图 8-1 显示了用电子表格对生成的数据进行显示。



可从
wrox.com
下载源代码

```

$ cat pci.sh
#!/bin/bash

getconfig()
{
    echo "PCI Devices,"
    lspci | sed s/" "/','/1
    echo "CPU Specification,"
    lscpu | sed s/" ":""/','/1 | tr -d ' '
}

echo -en "Getting system details..."
getconfig > pci.csv
echo "Done."
ls -l pci.csv

```

[pci.sh](#)

```

$ ./pci.sh
Getting system details...Done.
-rw-rw-r-- 1 steve steve 2297 Jan 24 21:14 pci.csv
$ oocalc pci.csv ←——— 这个命令启动 OpenOffice.org 电子表格软件。

```

[pci.csv](#)

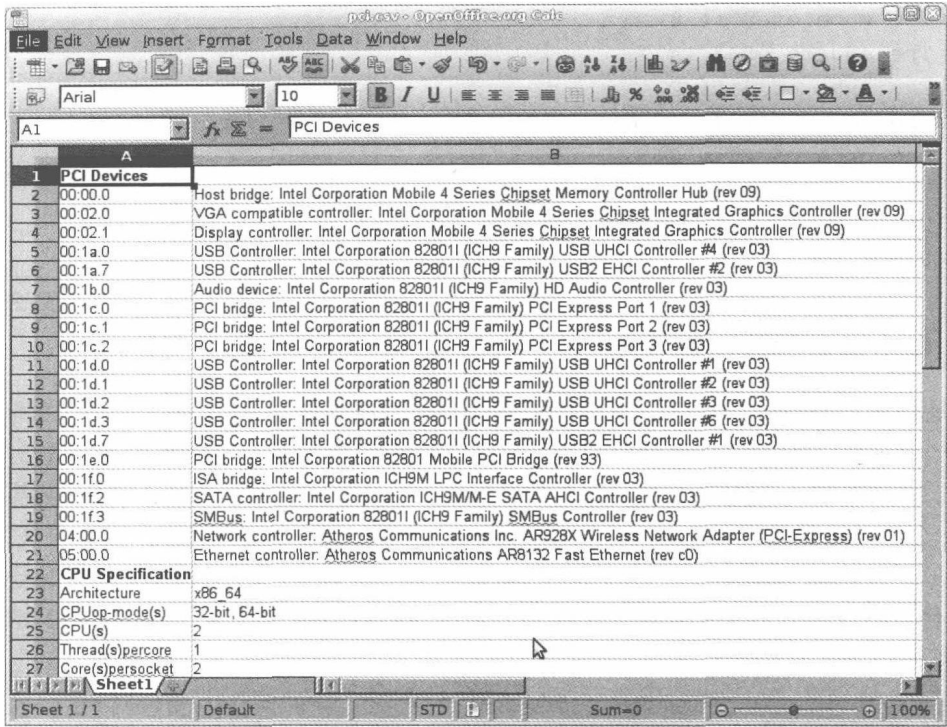



图8-1

调用者可能直接关心函数的输出，也可能将输出存储到文件中，而 `tee` 命令可以将输出同时导向 `stdout` 与文件。这在操作者需要看到输出的情况下非常有用，但仍然需要将输出存储到文件中，所以当脚本运行时需要一个日志文件来记录系统的状态。



可从
wrox.com
下载源代码

```
$ cat pid.sh
#!/bin/bash

process_exists()
{
    pidof "$1" && echo "These $1 process(es) were found." || \
        echo "No $1 processes were found."
}

echo "Checking system processes..."
process_exists apache2 | tee apache2.log
process_exists mysqld | tee mysql.log
process_exists squid | tee squid.log
$ ./pid.sh
Checking system processes...
2272 1452 1451 1450 1449 1448 1433
These apache2 process(es) were found.
No mysqld processes were found.
No squid processes were found.
$ cat apache2.log
```

```

2272 1452 1451 1450 1449 1448 1433
These apache2 process(es) were found.
$ cat mysql.log
No mysqld processes were found.
$ cat squid.log
No squid processes were found.
$

```

[pid.sh](#)

通常都会定义一个通用的日志函数，在脚本运行时用它接收诊断性的日志报告。这在任何脚本中都非常有用，而对于常见任务是运行多个命令(或命令集合)并记录结果的系统管理而言，这样的函数特别有用。为了让代码能说明问题，这里的输出特意弄得有些杂乱。该脚本不必担心输出是否美观，它只需要将结果记录下来并继续运行。脚本的作者可能没有时间精心编写脚本来将每条命令的结果优美地显示出来，但希望记录到底发生了什么。这些简单的函数实现了这一功能，且没有使主脚本过于复杂。像 `uname -a` 这样的命令，当添加到脚本名称、日期和时间后，会超出 80 个字符的列限制，但对于这样的日志文件而言并不重要。在本例中，单单 `uname -a` 就输出了 81 个字符，所以不是总能避免。



可从

wtox.com
下载源代码

```

$ cat debugger.sh
#!/bin/bash

LOGFILE=/tmp/myscript.log
# The higher the value of VERBOSE, the more talkative the log file is.
# Low values of VERBOSE mean a shorter log file;
# High values of VERBOSE mean a longer log file.
VERBOSE=10
APPNAME=`basename $0`

function logmsg()
{
    echo "${APPNAME}: `date`: $@" >> $LOGFILE
}

function debug()
{
    verbosity=$1
    shift
    if [ "$VERBOSE" -gt "$verbosity" ]; then
        echo "${APPNAME}: `date`: DEBUG Level ${verbosity}: $@" >> $LOGFILE
    fi
}

function die()
{
    echo "${APPNAME}: `date`: FATAL ERROR: $@" >> $LOGFILE
    exit 1
}

```

```

logmsg Starting script $0
uname -a || die uname command not found.
logmsg `uname -a`
cat /etc/redhat-release || debug 8 Not a RedHat-based system
cat /etc/debian_version || debug 8 Not a Debian-based system
cd /proc || debug 5 /proc filesystem not found.
grep -q "physical id" /proc/cpuinfo || debug 8 /proc/cpuinfo virtual
file not found.
logmsg Found `grep "physical id" /proc/cpuinfo | sort -u | wc -l`
physical CPUs.
unset IPADDR
./etc/sysconfig/network-scripts/ifcfg-eth0 || debug 1 ifcfg-eth0 not readable
logmsg eth0 IP address defined as $IPADDR
logmsg Script $0 finished.
$

$ ./debugger.sh
Linux goldie 2.6.32-5-amd64 #1 SMP Fri Dec 10 15:35:08 UTC 2010 x86_64 GNU/Linux
cat: /etc/redhat-release: No such file or directory
6.0
./debugger.sh: line 39: /etc/sysconfig/network-scripts/ifcfg-eth0:
No such file or Directory

```

debugger.sh

如果运行脚本并在/tmp/myscript.log 中查看输出,我们会发现这些不同的函数很容易地就为开发人员提供了标准的日志文件格式:

```

$ cat /tmp/myscript.log
debugger.sh: Mon Jan 24 23:46:40 GMT 2011: Starting script ./debugger.sh
debugger.sh: Mon Jan 24 23:46:40 GMT 2011: Linux goldie 2.6.32-5-amd64 #1
SMP Fri Dec 10 15:35:08 UTC 2010 x86_64 GNU/Linux
debugger.sh: Mon Jan 24 23:46:40 GMT 2011: DEBUG Level 8: Not a RedHat-based system
debugger.sh: Mon Jan 24 23:46:40 GMT 2011: Found 1 physical CPUs.
debugger.sh: Mon Jan 24 23:46:40 GMT 2011: DEBUG Level 1: ifcfg-eth0 not readable
debugger.sh: Mon Jan 24 23:46:40 GMT 2011: eth0 IP address defined as
debugger.sh: Mon Jan 24 23:46:40 GMT 2011: Script ./debugger.sh finished.
$

```



die()函数是 Perl 语言的一部分。对于编写临时应急脚本而言,这是一个值得借用的功能。

该脚本定义了 3 个函数——logmsg, 简单地记录传递给它的所有消息; debug, 如果脚本中的 VERBOSE 变量允许, 则记录传递给它的所有消息; die, 记录消息并终止脚本。脚本的主体随后尝试运行 3 种命令, 其中大多数都被记录下来, 而另一些则失败。注意下面这行命令:

```
logmsg Found `grep "physical id" /proc/cpuinfo | sort -u | wc -l` physical CPUs.
```

它显示所有传递给函数的参数，包括一些预定义的文本与系统命令的输出。对已安装的 CPU 的唯一物理 ID 进行计数得到总的物理 CPU 数目——随着多核芯片越来越常见，经常需要确定系统中已安装的实际芯片的数目。

8.1.5 函数陷阱

在 shell 脚本早期编写陷阱是很有用的。陷阱也可以采取其他形式，但最常见的用法是为脚本定义一个起清理作用的函数陷阱，在脚本被中断的时候调用。如果脚本使用了临时文件，则无论脚本由于什么原因被中断，都可以使用通用函数来删除它们。ldd.sh 这个脚本在目录中查找所有可以使用 ldd 找到链接库的文件。如果找到库文件，则让 ldd 显示这些文件，并发送返回码 0(成功)。

```
$ ldd /bin/ls
linux-vdso.so.1 => (0x00007fff573ff000)
libselinux.so.1 => /lib/libselinux.so.1 (0x00007f5c716a1000)
librt.so.1 => /lib/librt.so.1 (0x00007f5c71499000)
libacl.so.1 => /lib/libacl.so.1 (0x00007f5c71291000)
libc.so.6 => /lib/libc.so.6 (0x00007f5c70f30000)
libdl.so.2 => /lib/libdl.so.2 (0x00007f5c70d2c000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5c718d2000)
libpthread.so.0 => /lib/libpthread.so.0 (0x00007f5c70b0f000)
libattr.so.1 => /lib/libattr.so.1 (0x00007f5c7090b000)
$
```

像这样对文件系统的搜索可能花费很长的时间，所以应当允许脚本在运行完毕之前可能将其进程关闭。这意味着文件系统要不受影响，即使脚本被多次启动但从不运行完毕。该脚本确保其后续运行不会受到之前运行产生的旧的、未完成的文件的影响。



可从
wrox.com
下载源代码

```
$ cat ldd.sh
#!/bin/bash
# mktemp will give a pattern like "/tmp/tmp.U3XOAi92I2"
tempfile=`mktemp`
echo "Temporary file is ${tempfile}."
logfile=/tmp/libraries.txt
[ -f $logfile ] && rm -f $logfile

# Trap on:
# 1 = SIGHUP (Hangup of controlling terminal or death of parent)
# 2 = SIGINT (Interrupted by the keyboard)
# 3 = SIGQUIT (Quit signal from keyboard)
# 6 = SIGABRT (Aborted by abort(3))
# 9 = SIGKILL (Sent a kill command)

trap cleanup 1 2 3 6 9

function cleanup
```

```

{
    echo "Caught signal - tidying up..."
    # Tidy up after yourself
    rm -f ${tempfile}
    echo "Done. Exiting."
}

find $1 -type f -print | while read filename
do
    ldd ${filename} > ${tempfile}
    if [ "$?" -eq "0" ]; then
        let total=$total+1
        echo "File $filename uses libraries:" >> $logfile
        cat $tempfile >> $logfile
        echo >> $logfile
    fi
done
echo "Found `grep -c "^File " $logfile` files in $1 linked to libraries."
echo "Results in ${logfile}."

```

ldd.sh

```

$ ./ldd.sh /bin
Temporary file is /tmp/tmp.EYk2NnP6QW.
Found 78 files in /bin linked to libraries.
Results in /tmp/libraries.txt.
$ ls -l /tmp/tmp.EYk2NnP6QW          ls 命令显示临时文件已被删除。
ls: Cannot access /tmp/tmp.EYk2NnP6QW: No such file or directory.
$

```

再次运行该脚本，但在其完成之前按下[^]C，产生一个 SIGNIT 信号。尽管脚本没有结束，但 ls 命令再次确认了临时文件已被删除。/tmp/libraries.txt 文件不复存在，它只是被删除的临时文件。

```

$ ./ldd.sh /bin
Temporary file is /tmp/tmp.nhCOS9N3WE.
^CCaught signal - tidying up...
Done. Exiting.
Found 21 files in /bin linked to libraries.
Results in /tmp/libraries.txt.
$ ls -ld /tmp/tmp.nhCOS9N3WE
ls: /tmp/tmp.nhCOS9N3WE: No such file or directory
$ ls -ld /tmp/libraries.txt
-rw-rw-r-- 1 steve steve 16665 Mar 9 14:49 /tmp/libraries.txt
$

```

8.1.6 递归函数

递归函数是指将对自身的调用包含在执行过程中的函数。递归函数的一个简单例子

是数学中的阶乘函数。整数的阶乘是1到该整数之间所有整数的乘积。所以6的阶乘(也写成6!)是 $1 \times 2 \times 3 \times 4 \times 5 \times 6$ ，等于720。可以采用非递归的方法来计算阶乘，但因为 $6!$ 等于 $(5! \times 6)$ 、 $5!$ 等于 $(4! \times 5)$ ，依此类推，所以实际上可以将一个简单的阶乘函数写成下面这种形式：



```
$ cat factorial.sh
#!/bin/bash
factorial()
{
    if [ "$1" -gt "1" ]; then
        previous=`expr $1 - 1`
        parent=`factorial $previous`
        result=`expr $1 \* $parent`
        echo $result
    else
        echo 1
    fi
}

factorial $1
$ ./factorial.sh 6
720
```

factorial.sh

该脚本的含义不是很明显。增加一些日志记录与 `sleep` 调用(时间戳能更清晰地显示何时执行了什么命令)能有助于理解函数的输出，以及函数引起的副作用。函数 `factorial` 在 `echo` 语句之前被调用，而 `echo` 显示了运行中的结果。阶乘的计算在调试语句之前就完成了。日志的前6行说明 `factorial 6` 调用了 `factorial 5`，而 `factorial 5` 调用了 `factorial 4`，依此类推直到 11:50:12 时调用返回1的 `factorial 1`。日志随后说明函数正在退出递归，11:50:18 时调用 `factorial 2`，11:50:20 时调用 `factorial 3`，直到最后一步于 11:50:26 时调用 `factorial 6`。



```
$ cat recursive-string.sh
#!/bin/bash

LOG=/tmp/factorial-log.txt
> $LOG

factorial()
{
    echo "`date`: Called with $1" >> $LOG
    sleep 1
    if [ "$1" -gt "1" ]; then
        previous=`expr $1 - 1`
        parent=`factorial $previous`
        result=`expr $1 \* $parent`
        echo "`date`: Passed $1 - the factorial of $previous is ${parent}. \"$1 * $parent is ${result}." >> $LOG
    fi
}
```

```
    echo "`date`: Sleeping for 2 seconds." >> $LOG
    sleep 2
    echo $result
else
    echo "`date`: Passed $1 - returning 1." >> $LOG
    echo "`date`: Sleeping for 5 seconds." >> $LOG
    sleep 5
    echo 1
fi
}

read -p "Enter a number: " x
echo "Started at `date`"
factorial $x
echo "Here is my working:"
cat $LOG
rm -f $LOG
echo "Finished at `date`"
$ ./recursive-string.sh
Enter a number: 6
Started at Wed Jan 26 11:50:07 GMT 2011
720
Here is my working:
Wed Jan 26 11:50:07 GMT 2011: Called with 6
Wed Jan 26 11:50:08 GMT 2011: Called with 5
Wed Jan 26 11:50:09 GMT 2011: Called with 4
Wed Jan 26 11:50:10 GMT 2011: Called with 3
Wed Jan 26 11:50:11 GMT 2011: Called with 2
Wed Jan 26 11:50:12 GMT 2011: Called with 1
Wed Jan 26 11:50:13 GMT 2011: Passed 1 - returning 1.
Wed Jan 26 11:50:13 GMT 2011: Sleeping for 5 seconds.
Wed Jan 26 11:50:18 GMT 2011: Passed 2 - the factorial of 1 is 1.
2 * 1 is 2.
Wed Jan 26 11:50:18 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:50:20 GMT 2011: Passed 3 - the factorial of 2 is 2.
3 * 2 is 6.
Wed Jan 26 11:50:20 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:50:22 GMT 2011: Passed 4 - the factorial of 3 is 6.
4 * 6 is 24.
Wed Jan 26 11:50:22 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:50:24 GMT 2011: Passed 5 - the factorial of 4 is 24.
5 * 24 is 120.
Wed Jan 26 11:50:24 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:50:26 GMT 2011: Passed 6 - the factorial of 5 is 120.
6 * 120 is 720.
Wed Jan 26 11:50:26 GMT 2011: Sleeping for 2 seconds.
Finished at Wed Jan 26 11:50:28 GMT 2011
$
```

recursive-string.sh

上面的脚本名为 `recursive-string.sh`，因为它以字符串的形式传递数字。接下来的脚本名为 `recursive-byte.sh`，因为它使用来自函数的返回值来返回实际计算出的数字。因为 Unix 返回码是字节，所以只能表示 0~255 之间的值。我们看到该脚本对于输入大到 5 时还能正确计算阶乘，但如果输入为 6 时则失败，因为 6 的阶乘为 $6*120=720$ ，超过了 255。



可从
WTOX.COM
下载源代码

```
$ cat recursive-byte.sh
```

```
#!/bin/bash
```

```
LOG=/tmp/factorial-log.txt
```

```
> $LOG
```

```
factorial()
```

```
{
```

```
    echo "`date`: Called with $1" >> $LOG
```

```
    sleep 1
```

```
    if [ "$1" -gt "1" ]; then
```

```
        previous=`expr $1 - 1`
```

```
        factorial $previous
```

```
        parent=$?
```

```
        result=`expr $1 \* $parent`
```

```
        echo "`date`: Passed $1 - the factorial of $previous is ${parent}. " \
```

```
            "$1 * $parent is ${result}." >> $LOG
```

```
        echo "`date`: Sleeping for 2 seconds." >> $LOG
```

```
        sleep 2
```

```
        return $result
```

```
    else
```

```
        echo "`date`: Passed $1 - returning 1." >> $LOG
```

```
        echo "`date`: Sleeping for 5 seconds." >> $LOG
```

```
        sleep 5
```

```
        return 1
```

```
    fi
```

```
}
```

```
read -p "Enter a number: " x
```

```
echo "Started at `date`"
```

```
factorial $x
```

```
echo "Answer: $?"
```

```
echo "Here is my working:"
```

```
cat $LOG
```

```
rm -f $LOG
```

```
echo "Finished at `date`"
```

```
$ ./recursive-byte.sh
```

```
Enter a number: 5
```

```
Started at Wed Jan 26 11:53:49 GMT 2011
```

```
Answer: 120
```

```
Here is my working:
```

```
Wed Jan 26 11:53:49 GMT 2011: Called with 5
```

```
Wed Jan 26 11:53:50 GMT 2011: Called with 4
```

```
Wed Jan 26 11:53:51 GMT 2011: Called with 3
```



```

Wed Jan 26 11:53:52 GMT 2011: Called with 2
Wed Jan 26 11:53:53 GMT 2011: Called with 1
Wed Jan 26 11:53:54 GMT 2011: Passed 1 - returning 1.
Wed Jan 26 11:53:54 GMT 2011: Sleeping for 5 seconds.
Wed Jan 26 11:53:59 GMT 2011: Passed 2 - the factorial of 1 is 1.
2 * 1 is 2.
Wed Jan 26 11:53:59 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:54:01 GMT 2011: Passed 3 - the factorial of 1 is 2.
3 * 2 is 6.
Wed Jan 26 11:54:01 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:54:03 GMT 2011: Passed 4 - the factorial of 1 is 6.
4 * 6 is 24.
Wed Jan 26 11:54:03 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:54:05 GMT 2011: Passed 5 - the factorial of 1 is 24.
5 * 24 is 120.
Wed Jan 26 11:54:05 GMT 2011: Sleeping for 2 seconds.
Finished at Wed Jan 26 11:54:07 GMT 2011
$

```

recursive-byte.sh

在向该脚本传递数字 6 时，得到 208，因为 720 模 256 等于 208。为了显示结果，程序 `bc` 可以得到正确的答案，但当 `shell` 函数接收返回值时，再次将其压缩成单个字节，并且 720 的有效位就是 208。



因为一个字节由 8 个位组成，所以超过 255 的数会回绕。用二进制表示的话，120 是 1111000，包含 8 个位，可以正确表示。而 720 是 1011010000，包含 10 个位。其前两个位被截去，剩下 11010000，十进制表示为 208。

```

$ ./recursive-byte.sh
Enter a number: 6
Started at Wed Jan 26 11:54:46 GMT 2011
Answer: 208
Here is my working:
Wed Jan 26 11:54:46 GMT 2011: Called with 6
Wed Jan 26 11:54:47 GMT 2011: Called with 5
Wed Jan 26 11:54:48 GMT 2011: Called with 4
Wed Jan 26 11:54:49 GMT 2011: Called with 3
Wed Jan 26 11:54:50 GMT 2011: Called with 2
Wed Jan 26 11:54:51 GMT 2011: Called with 1
Wed Jan 26 11:54:52 GMT 2011: Passed 1 - returning 1.
Wed Jan 26 11:54:52 GMT 2011: Sleeping for 5 seconds.
Wed Jan 26 11:54:57 GMT 2011: Passed 2 - the factorial of 1 is 1. 2 * 1 is 2.
Wed Jan 26 11:54:57 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:54:59 GMT 2011: Passed 3 - the factorial of 1 is 2. 3 * 2 is 6.
Wed Jan 26 11:54:59 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:55:01 GMT 2011: Passed 4 - the factorial of 1 is 6. 4 * 6 is 24.
Wed Jan 26 11:55:01 GMT 2011: Sleeping for 2 seconds.

```

```

Wed Jan 26 11:55:03 GMT 2011: Passed 5 - the factorial of 1 is 24. 5 * 24 is 120.
Wed Jan 26 11:55:03 GMT 2011: Sleeping for 2 seconds.
Wed Jan 26 11:55:05 GMT 2011: Passed 6 - the factorial of 1 is 120. 6 * 120 is 720
.
Wed Jan 26 11:55:05 GMT 2011: Sleeping for 2 seconds.
Finished at Wed Jan 26 11:55:07 GMT 2011
$

```

如果使用得当，递归是一项非常灵活的技术。对于递归的核心理解在于知道何时进行调用，以及循环如何退出。另外很重要的一点是不要过度使用资源——如果递归函数的每次调用实例都要打开一个文件，则打开的文件数目可能超过系统的允许范围。

8.2 变量的作用域

函数一般只返回单个值。`shell` 则不会像函数一样简单，因为偶尔会出现一些副作用。只有一种 `shell` 在执行脚本之后，函数中的所有变量依然存在。该脚本有 3 个变量，`GLOBALVAR`、`myvar` 和 `uniquevar`。在一些语言中，`GLOBALVAR` 被当成类似环境中的全局变量，因为它被定义在函数本身以外。而 `myvar` 是函数中的局部变量，并且独立于函数外的 `myvar` 变量。在很多语言中，在第一次使用 `uniquevar` 时都会失败，因为它在使用之前没有被定义。而在 `shell` 中，这是可以的。`uniquevar` 在函数之外是不可见的，因为它不存在于函数之外。

这些有点冲突却又被广泛理解的标准在很多编程语言中都很常见。`shell` 有些不同，因为它不是严格的编程语言。在运行下面的脚本之前试着预测一下它的结果，并解释为何结果与预期的一致。可以使用表 8-1。

表 8-1 `GLOBALVAR`、`myvar`和`uniquevar` 的预测结果

变 量	初 始 值	预测结果与理由
<code>\$GLOBALVAR</code>	1	
<code>\$myvar</code>	500	
<code>\$uniquevar</code>	未定义	



可从
wrox.com
下载源代码

```

$ cat scope.sh
#!/bin/sh

the_function()
{
    echo " This is the_function. I was passed $# arguments: \"\
        \"\$1\", \"\$2\", \"\$3\"."
    myvar=100
    echo "    GLOBALVAR started as $GLOBALVAR;"
    GLOBALVAR=`expr $GLOBALVAR + 1`
    echo "    GLOBALVAR is now $GLOBALVAR"
    echo "    myvar started as $myvar;"
}

```

```

myvar=`expr $myvar + 1`
echo "    myvar is now $myvar"
echo "uniquevar started as $uniquevar"
uniquevar=`expr $uniquevar + 1`
echo "uniquevar is $uniquevar"
echo "    Leaving the_function."
}

GLOBALVAR=1
myvar=500
echo "This is the main script."
echo "GLOBALVAR is $GLOBALVAR"
echo "myvar is $myvar"
echo "I was passed $# arguments: \"\
    \"$1\", \"$2\", \"$3\"."
echo "*****"
echo

echo "Calling the_function with 1 2 3 ..."
the_function 1 2 3
echo "GLOBALVAR is $GLOBALVAR"
echo "myvar is $myvar"
echo
echo "Calling the_function with 3 2 1 ..."
the_function 3 2 1
echo "GLOBALVAR is $GLOBALVAR"
echo "myvar is $myvar"
echo
echo "Calling the_function with $1 $2 $3 ..."
the_function $1 $2 $3
echo "GLOBALVAR is $GLOBALVAR"
echo "myvar is $myvar"
echo
echo "All Done."
echo "GLOBALVAR is $GLOBALVAR"
echo "myvar is $myvar"
echo "uniquevar is $uniquevar"
$

```

scope.sh

首先，GLOBALVAR 是函数调用时传递到函数 `the_function` 的环境的一部分，所以 `the_function` 应当有 GLOBALVAR 的初始值。因为函数调用处于一个 shell 中，所以 GLOBALVAR 像任何全局变量一样被增加，并且在后续调用 `the_function` 时，GLOBALVAR 再次被增加。

其次，myvar 与 GLOBALVAR 不同，因为它在 `the_function` 函数中被定义。这对于很多编程语言而言是指此处的 myvar 与别处的不同。然而这并不适用于 shell。如果 myvar 被赋过值，则对于运行的进程(此处就是 shell)而言，它就是变量的值，并且无论进程本身在函数中或处于其他状态。myvar 是一个简单的 Unix 系统变量，没有函数、shell 或作用域的

概念。

最后, `uniquevar` 具有最明显的潜在性问题, 因为它没有经过定义。这不会有什么问题。`shell` 会将其处理为空字符串。在第一次引用时为 `uniquevar='expr $uniquevar + 1'`。传递给 `expr` 的是 `uniquevar='expr + 1'`, 并且 `expr` 会将其计算为 1。此时, `uniquevar` 为 1, 并且下次引用时增加到 2 等。`uniquevar` 在函数之外还保留其值, 并在主脚本的末尾进行回显。

```
$ ./scope.sh a b c
This is the main script.
GLOBALVAR is 1
myvar is 500
I was passed 3 arguments: "a", "b", "c".
*****

Calling the_function with 1 2 3 ...
  This is the_function. I was passed 3 arguments: "1", "2", "3".
    GLOBALVAR started as 1;
    GLOBALVAR is now 2
    myvar started as 100;
    myvar is now 101
  uniquevar started as
  uniquevar is 1
    Leaving the_function.
  GLOBALVAR is 2
  myvar is 101

Calling the_function with 3 2 1 ...
  This is the_function. I was passed 3 arguments: "3", "2", "1".
    GLOBALVAR started as 2;
    GLOBALVAR is now 3
    myvar started as 100;
    myvar is now 101
  uniquevar started as 1
  uniquevar is 2
    Leaving the_function.
  GLOBALVAR is 3
  myvar is 101

Calling the_function with a b c ...
  This is the_function. I was passed 3 arguments: "a", "b", "c".
    GLOBALVAR started as 3;
    GLOBALVAR is now 4
    myvar started as 100;
    myvar is now 101
  uniquevar started as 2
  uniquevar is 3
    Leaving the_function.
  GLOBALVAR is 4
  myvar is 101
```

```
All Done.
GLOBALVAR is 4
myvar is 101
uniquevar is 3
$
```

`myvar` 被当成全局变量可能是一种特殊情况。函数经常需要使用自己的变量而又不会覆盖调用者使用的变量值,而且函数对于调用者一无所知。将局部变量命名为 `the_function_variable_one`、`the_function_variable_two` 等,或者 `the_function_stock` 与 `the_function_price`,这样很快就会显得笨拙不堪。`bash shell` 通过增加关键字 `local` 来处理这种情况。当向函数增加一行 `local myvar` 时,结果会不同。



可从
wrox.com
下载源代码

```
the_function()
{
    echo "  This is the_function. I was passed $# arguments: "\
        "\"$1\", \"$2\", \"$3\"."
    local myvar=100
    echo "    GLOBALVAR started as $GLOBALVAR;"
    GLOBALVAR=`expr $GLOBALVAR + 1`
    echo "    GLOBALVAR is now $GLOBALVAR"
    echo "    myvar started as $myvar;"
    myvar=`expr $myvar + 1`
    echo "    myvar is now $myvar"
    echo "uniquevar started as $uniquevar"
    uniquevar=`expr $uniquevar + 1`
    echo "uniquevar is $uniquevar"
    echo "  Leaving the_function."
}
```

scope2.sh

这一额外的关键字告诉 `bash`, `myvar` 与外部的同名变量是不同的。`the_function` 中对 `myvar` 的任何操作都不会影响到另一个也叫 `myvar` 的变量。也就是说, `the_function` 可以安全地使用任何变量名。下面的脚本 `scope2.sh` 显示了将 `myvar` 声明为 `local` 的不同之处。



显然,如果 `the_function` 需要引用另一个 `myvar` 变量的值,则必须为局部变量起一个不同的名称。

```
$ ./scope2.sh a b c
This is the main script.
GLOBALVAR is 1
myvar is 500
I was passed 3 arguments: "a", "b", "c".
*****

Calling the_function with 1 2 3 ...
This is the_function. I was passed 3 arguments: "1", "2", "3".
```

```

    GLOBALVAR started as 1;
    GLOBALVAR is now 2
    myvar started as 100;
    myvar is now 101
uniquevar started as
uniquevar is 1
    Leaving the_function.
GLOBALVAR is 2
myvar is 500

Calling the_function with 3 2 1 ...
    This is the_function. I was passed 3 arguments: "3", "2", "1".
    GLOBALVAR started as 2;
    GLOBALVAR is now 3
    myvar started as 100;
    myvar is now 101
uniquevar started as 1
uniquevar is 2
    Leaving the_function.
GLOBALVAR is 3
myvar is 500

Calling the_function with a b c ...
    This is the_function. I was passed 3 arguments: "a", "b", "c".
    GLOBALVAR started as 3;
    GLOBALVAR is now 4
    myvar started as 100;
    myvar is now 101
uniquevar started as 2
uniquevar is 3
    Leaving the_function.
GLOBALVAR is 4
myvar is 500

All Done.
GLOBALVAR is 4
myvar is 500
uniquevar is 3
$

```

额外的 `local` 关键字使得 `bash` 成为可以编写更复杂的函数与库的实用 `shell`。函数可以修改任意变量的值，像这样的标准行为也应当被清晰地定义，否则调试会非常困难。

使用函数可以重复执行代码，而不用重复编写或者编写作为独立 `shell` 脚本的每个代码块。还可以用函数建立由相关函数组成的库。

8.3 库

`shell` 没有 Perl 与 C 所使用的真正意义上的库。在 C 语言中，我们可以通过包含头文

件与链接到相应的库(简称为 **m**，因此 **gcc** 调用中使用 **-lm** 进行链接)来使用数学库。然后程序可以使用一些新增的函数，包括 **cos()**、**sin()** 和 **tan()**。



可从
wrox.com
下载源代码

```
$ cat math.c
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
    int arg=atoi(argv[1]);
    printf("cos(%d)=%0.8f\n", arg, cos(arg));
    printf("sin(%d)=%0.8f\n", arg, sin(arg));
    printf("tan(%d)=%0.8f\n", arg, tan(arg));
    return 0;
}

$ gcc -lm -o math math.c
$ ./math 30
cos(30)=0.15425145
sin(30)=-0.98803162
tan(30)=-6.40533120
$ ./math 60
cos(60)=-0.95241298
sin(60)=-0.30481062
tan(60)=0.32004039
$ ./math 90
cos(90)=-0.44807362
sin(90)=0.89399666
tan(90)=-1.99520041
$
```

math.c



这一小段 C 代码没有对输入进行合理性测试，而只是在此处展示如何在 C 语言中链接一个库。另外，如果看起来结果有误，那是因为它处理的是弧度而不是角度。

shell 有不同的定义标准设置的方法——别名、变量和函数，它们创建的环境与库集合几乎一样。当 **shell** 被非交互式调用时，它会读取 **~/.profile**，然后读取 **~/.bashrc**、**~/.kshrc** 或类似文件(取决于具体使用的 **shell**)。我们的脚本也可以使用同样的方法调用(**source**)其他文件，获取它们的内容，而不用运行任何实际代码。这些文件中定义的函数也能供我们使用。



与 **~/.bashrc** 进行一些定义操作但实际上并不执行任何代码一样，库在运行期间不会执行任何代码一样，而是读取其中定义的函数。

如果 shell 脚本可以读取用户的变量集合以及最有用的函数,那么就能利用一个自定义的环境并获得本章开头提到的模块化带来的好处。

按照这样的逻辑可以得到一个结论, shell 脚本函数库可以在用户环境中定义,并能用于所有 shell 脚本与交互式会话,或者是用于某个特殊脚本的库集合。

8.3.1 库的创建与访问

除了没有实际的运行起始点外,库的创建方法与 shell 脚本一样——所有对库要做的就是定义函数。实际调用在主 shell 脚本中。另外,函数可能调用同一库中的其他函数,甚至其他库中的函数。



库中的函数不能调用同一库中后面定义的函数,这似乎是理所当然的。然而,因为函数要稍后才会被执行,所以只要代码语法正确,先定义的函数调用后定义的函数是可以的。因为主脚本调用这些函数时,它们已经定义在环境中了。这与脚本调用一个不存在的二进制文件是一样的。语法是正确的,所以 shell 能正确地进行分析。

shell 脚本文件名通常以.sh 结尾(甚至是.bash、.ksh 等,只要有助于表示脚本需要某个特定 shell)。库不必标记为可执行,并且通常都没有任何扩展名。库也不应当以#!/开头,因为它们不会被操作系统执行,而只是被 shell 本身读取。

将库包含在 shell 脚本中的方法是使用或者 source 命令来调用库文件名。如果设置了 shell 选项 shopt sourcepath,则 bash 将搜索\$PATH 中同名的文件,然后在当前目录下搜索。否则,脚本只在当前目录下搜索。source 命令将文件中的所有内容读取到当前环境,所以库中任何函数定义都可供当前正在执行的 shell 使用。类似地,库中定义的任何变量也会在当前 shell 中进行设置。

8.3.2 库的结构

在编写较大的脚本集合时,最好建立一个 lib/目录来存放希望经常使用的那些函数。这样做的好处是本章开头提到的一致性、可读性与可维护性,以及从多个脚本中调用相同函数的可重用性。对于通用的库,\$HOME/lib 是一个比较合适的位置。对于/usr/local/myapp/中的脚本,/usr/local/myapp/bin 与/usr/local/myapp/lib 这两个位置则更加适合。

库可以相互调用,并且继承关系按预期的方式起作用:如果 lib1 包含了 lib2,则任何包含了 lib1 的脚本或库也会继承 lib2 中的内容。



可从
wrox.com
下载源代码

```
$ cat calling.sh
#!/bin/sh

. ./lib1

func1
```



```
echo "I also get func2 for free..."
func2
```

calling.sh

```
$ cat lib1
. ./lib2
```

```
func1()
{
    echo func1
    func2
}

anotherfunc()
{
    echo More stuff here.
}
```

lib1

```
$ cat lib2
func2()
{
    echo func2
}
```

lib2

```
$ ./calling.sh
func1
func2
I also get func2 for free...
func2
$
```

然而，这些库之间不能完全相互包含，否则会得到错误消息 `too many open files`，因为 `shell` 不停地对文件进行递归打开，直到耗尽所有资源。让 `func2` 调用 `anotherfunc` 要求 `lib2` 包含 `anotherfunc` 的定义(尽管 `calling.sh` 脚本现在知道所有的函数定义，但是 `lib2` 本身不知道)，也就是说 `lib2` 必须包含 `lib1`。因为 `lib1` 也包含 `lib2`，结果 `shell` 陷入一个递归循环，导致下面的错误：



任何时刻能打开的文件的最大数目由 `/etc/security/limits.conf` 中的 `nofile` 参数定义。

```
$ ./calling.sh
.: 1: 3: Too many open files
$
```

一个解决方法是让 `calling.sh` 包含两个库。这样，在调用的时候，`lib2` 将知道 `anotherfunc` 的定义。这确实很奏效。只要两个库文件不相互包含，并且调用脚本包含任何所需的库文件，那么整个环境就能正确地进行设置。也就是说，`calling.sh` 必须知道任何对于库结构的修改。这可以通过添加一个元库文件来规避。此处的元库文件是 `lib`：



```
$ cat calling.sh
```

```
#!/bin/sh
```

```
.. ./lib
```

```
func1
```

```
echo "I also get func2 for free..."
```

```
func2
```

```
$ cat lib
```

```
.. ./lib1
```

```
.. ./lib2
```

lib

```
$ ./calling.sh
```

```
func1
```

```
func2
```

```
More stuff here.
```

```
I also get func2 for free...
```

```
func2
```

```
More stuff here.
```

```
$
```

还可以模仿 C 语言的头文件结构来更加灵活地管理这类相互依赖的关系，因为它与 C 语言中的这类问题非常相似。在 C 语言中，`stdio.h` 这样的头文件首先检查 `_STDIO_H` 是否已被定义。如果已经定义，则表示 `stdio.h` 已经被包含；否则，`stdio.h` 会声明其中包含的全部内容。

```
#ifndef _STDIO_H
```

```
# define _STDIO_H 1
```

```
... stdio declarations go here ...
```

```
#endif
```

shell 能做类似的事情，相比 C 语言甚至更加美观。通过定义一个与库名称相同的变量，shell 能判断库是否已经被包含到环境中。这样一来，如果多个库试图进行相互引用，shell 不会陷入无限循环。



注意, 下面的例子同样模仿了 C 语言中用下划线作为系统(或库)变量名前缀的惯例。以下划线开头的变量与任何其他变量并没有技术上的差异, 但可以突显出它是作为系统变量或元变量, 而不会与存储用户数据的变量混淆。当这些例库被称为 lib1 和 lib2 时, 看似不会造成任何问题。如果脚本要处理库存管理这样的事务, 可能有一个称为 stock 的库与另一个称为 store 的库。它们看上去就应当作为表示数据的变量, 所以按照下划线的惯例, 我们就可以避免元数据变量 `$_stock` 和 `$_store` 与实际数据变量 `$stock` 和 `$store` 出现变量名重合。

```
$ cat calling.sh
#!/bin/sh

. ./lib1

func1

echo "I also get func2 for free..."
func2
$ cat lib1
_lib1=1
[ -z "$_lib2" ] && . ./lib2

func1()
{
    echo func1
    func2
}

anotherfunc()
{
    echo More stuff here.
}
$ cat lib2
_lib2=1
[ -z "$_lib1" ] && . ./lib1

func2()
{
    echo func2
    anotherfunc
}
$ ./calling.sh
func1
func2
More stuff here.
I also get func2 for free...
func2
More stuff here.
$
calling.sh
```

这个解决方案更加健壮。如果可以的话，请尽量使用单个库文件或者减小库之间的依赖关系；如果无法做到，请按照上面的方法设置标志变量控制这些依赖关系。

8.3.3 网络配置库

我们已经介绍了库的工作原理，现在开始编写一个实实在在脚本。该脚本包含一个中心脚本用来对新的网络接口进行配置。它依赖于一个库，这个库又决定了脚本所在特定操作系统的任务实现方法。注意，只要中心脚本做到足够通用化，这样的结构就能进行较大的扩展，从而得到一个可扩展、可维护的跨平台 shell 脚本库。该库包含 4 个文件与 1 个 shell 脚本：

- network.sh
- definitions
- debian-network
- redhat-network
- solaris-network

前两个文件定义了库的基本结构：通用化的定义在 definitions 文件中，而 network.sh 是顶层脚本。



可从
wrox.com
下载源代码

```
$ cat definitions
# Various error conditions. It is better to
# provide generic definitions so that the
# individual libraries are that much clearer.
_WRONG_PLATFORM=1
_NO_IP=2
_NO_CONFIG=3

# Success is a variant on failure - best to define this too for consistency.
SUCCESS=0
$
```

definitions

```
$ cat network.sh
#!/bin/bash

[ -z "$_definitions" ] && . definitions
[ -f /etc/redhat-release ] && . ./redhat-network
[ -f /etc/debian_version ] && . ./debian-network
[ `uname` == "SunOS" ] && . ./solaris-network

for thisnic in $*
do
    thisip=`getipaddr $thisnic`
    case $? in
        $SUCCESS) echo "The IP Address configured for $thisnic is $thisip" ;;
        $_WRONG_PLATFORM) echo "This does not seem to be running on the
```

```

expected platform" ;;
    $_NO_IP) echo "No IP Address defined for $thisnic" ;;
    $_NO_CONFIG) echo "No configuration found for $thisnic" ;;
esac
done
$

```

network.sh

还有 3 个不同的库文件，分别包含了 `getipaddr` 的定义，但只有适当的定义实例会出现在运行系统中。每个系统都有完全不同的 `getipaddr` 实现方法，但是库并不关心这一点。



可从
wrox.com
下载源代码

```

$ cat redhat-network
[ -z "$_definitions" ] && . definitions

# RedHat-specific getipaddr() definition
getipaddr()
{
    [ -d /etc/sysconfig/network-scripts ] || return _$WRONG_PLATFORM
    if [ -f /etc/sysconfig/network-scripts/ifcfg-$1 ]; then
        unset IPADDR
        . /etc/sysconfig/network-scripts/ifcfg-$1
        if [ -z "$IPADDR" ]; then
            return $_NO_IP
        else
            echo $IPADDR
        fi
    else
        return $_NO_CONFIG
    fi
    # Not strictly needed
    return $SUCCESS
}
$

```

redhat-network

```

$ cat debian-network
[ -z "$_definitions" ] && . ./definitions

# Debian-specific getipaddr() definition
getipaddr()
{
    [ -f /etc/network/interfaces ] || return $_WRONG_PLATFORM
    found=0
    while read keyword argument morestuff
    do
        #echo "Debug: k $keyword a $argument m $morestuff"
        if [ "$keyword" == "iface" ]; then
            if [ "$found" -eq "1" ]; then
                # we had already found ours, but no address line found.

```

```

        return $_NO_IP
    else
        if [ "$argument" == "$1" ]; then
            found=1
        fi
    fi
fi
if [ "$found" -eq "1" ]; then
    if [ "$keyword" == "address" ]; then
        # Found the address of this interface.
        echo $argument
        return $SUCCESS
    fi
fi
done < /etc/network/interfaces
if [ "$found" -eq "0" ]; then
    return $_NO_CONFIG
fi
# Not strictly needed
return $SUCCESS
}

```

debian-network

```

$ cat solaris-network
[ -z "$_definitions" ] && . ./definitions

# Solaris-specific getipaddr() definition
getipaddr()
{
    uname | grep SunOS > /dev/null || return $_WRONG_PLATFORM
    [ -f /etc/hostname.${1} ] || return $_NO_CONFIG
    [ ! -s /etc/hostname.${1} ] && return $_NO_IP
    getent hosts `head -1 /etc/hostname.${1} | cut -d"/" -f1 | \
        awk '{ print $1 }'` | cut -f1 || cat /etc/hostname.${1}
    return $SUCCESS
}
$

```

solaris-network

主脚本可以自由地调用库中的函数，而不必知晓不同操作系统之间的区别或是库处理这些区别的方法。主脚本只是简单地调用 `getipaddr`，然后得到所需的返回结果。在只定义了 `eth0` 的 Debian 系统中，运行结果如下：

```

debian# ./network.sh eth0 eth1 bond0 bond1 eri0 qfe0 wlan0
The IP Address configured for eth0 is 192.168.1.13
No configuration found for eth1
No configuration found for bond0
No configuration found for bond1

```

```
No configuration found for eri0
No configuration found for qfe0
No configuration found for wlan0
debian#
```

在配置了 eth0、eth1 和 bond0，但 eth0 没有 IP 地址(bond0 的一部分)的 Red Hat 系统中，运行结果如下：

```
redhat# ./network.sh eth0 eth1 bond0 bond1 eri0 qfe0 wlan0
No IP Address defined for eth0
The IP Address configured for eth1 is 192.168.44.107
The IP Address configured for bond0 is 192.168.81.64
No configuration found for bond1
No configuration found for eri0
No configuration found for qfe0
No configuration found for wlan0
redhat#
```

在配置了 eri0 和 qfe0 的 Solaris 系统中，运行结果如下：

```
solaris# ./network.sh eth0 eth1 bond0 bond1 eri0 qfe0 wlan0
No configuration found for eth0
No configuration found for eth1
No configuration found for bond0
No configuration found for bond1
The IP Address configured for eri0 is 192.168.101.3
The IP Address configured for qfe0 is 192.168.190.45
No configuration found for wlan0
solaris#
```

这样使得一个简单的接口——getipaddr 被调用，而不需要调用脚本知道与实现相关的内容。最后将 network.sh 本身做成一个库，这样调用脚本甚至不必知道 redhat-network、debian-network 与 solaris-network 库的存在。调用脚本只需要包含 network 库并调用 shownetwork 来显示网络信息。\$*在函数中的工作原理说明不需要对代码进行修改便能将其从脚本迁移到函数，反之亦然。



可从
wrox.com
下载源代码

```
$ cat network
```

```
[ -f /etc/redhat-release ] && . ./redhat-network
[ -f /etc/debian_version ] && . ./debian-network
[ `uname` == "SunOS" ] && . ./solaris-network
```

```
shownetwork()
{
    for thisnic in $*
    do
        thisip=`getipaddr $thisnic`
        case $? in
            $SUCCESS) echo "The IP Address configured for $thisnic is $thisip" ;;
```

```

$_WRONG_PLATFORM) echo "This does not seem to be running " \
    "on the expected platform" ;;
$_NO_IP) echo "No IP Address defined for $thisnic" ;;
$_NO_CONFIG) echo "No configuration found for $thisnic" ;;
esac
done
}
$

```

network

```

$ cat client.sh
#!/bin/bash

. ./network

shownetwork $@

```

client.sh

```

$ ./client.sh eth0
The IP Address configured for eth0 is 192.168.1.13
$

```

8.3.4 库的使用

库非常有用。很多系统管理员收集脚本、函数与别名来形成他们各自的工具箱，好让日常的管理工作更加方便与高效。在较复杂的脚本中使用与任务相关的库也能够让开发、调试与维护更加简单。在团队之间共享库是一种很好的知识共享的方式，但实际上大多数管理员倾向于使用由团队维护的更具健壮性的共享工具集来独立维护各自的工具集。这有助于标准的正式化，加快常规任务的发布并能确保发布的一致性。

8.4 getopts

函数、脚本与其他程序都以相同的方式传递参数——\$1 表示程序、脚本或函数的第一个参数。调用脚本很难判断它调用的是外部的程序、脚本还是函数。所以让函数接收参数的方式与脚本一致是有意义的。

mkfile 脚本是对 dd 的封装，而 dd 提供的语法与 Unix 中二进制程序 mkfile 相似。该脚本在主脚本中使用 getopts 对参数进行分析，展示了 getopts 的基本用法。该脚本的合法选项如下：

- -i infile (对输入进行复制的文件名，默认为/dev/zero)
- -b blocksize (每个读写块的大小，默认为 512KB)
- -q 表示脚本静默执行
- -? 显示用法信息

对于任何其他参数输入,脚本都会显示用法信息。`-i`与`-b`选项都需要一个参数;`-q`与`-?`则不需要。通过 `getopts` 语句`'i:b:q?'`可以简洁地把选项与参数的信息表示出来。冒号(:)表示所需的一个参数,所以 `i:`与 `b:`表示`-i`与`-b`各需要一个参数,并且若没有提供参数则会出错。另外两个字符后面没有冒号,所以它们是没有参数的独立标志。当传递参数时,参数值被置入到变量 `OPTARG` 中。



尽管本书中大多数 shell 脚本都带有文件名扩展`.sh`,但 `mkfile` 是个已存在的二进制文件,而且该脚本为其提供了一个通用型的功能。因此,该脚本实例与二进制 `mkfile` 具有相同的名称是有意义的。



可从
wrox.com
下载源代码

```
$ cat mkfile
#!/bin/bash
# wrapper for dd to act like Solaris' mkfile utility.

function usage()
{
    echo "Usage: mkfile [ -i infile ] [ -q ] [ -b blocksize ] size[k|m|g] filename"
    echo "Blocksize is 512 bytes by default."
    exit 2
}

function humanreadable ()
{
    multiplier=1
    case $1 in
        *b) multiplier=1 ;;
        *k) multiplier=1024 ;;
        *m) multiplier=1048576 ;;
        *g) multiplier=1073741824 ;;
    esac
    numeric=`echo $1 | tr -d 'k'|tr -d 'm'|tr -d 'g'|tr -d 'b'`
    expr $numeric \* $multiplier
}

# mkfile uses 512 byte blocks by default - so shall I.
bs=512
quiet=0
INFILE=/dev/zero

while getopts 'i:b:q?' argv
do
    case $argv in
        i) INFILE=$OPTARG ;;
        b) bs=$OPTARG ;;
        q) quiet=1 ;;
        \?) usage ;;
    esac
done
```

```

done

for i in `seq 2 ${OPTIND}`
do
    shift
done

if [ -z "$1" ]; then
    echo "ERROR: No size specified"
fi
if [ -z "$2" ]; then
    echo "ERROR: No filename specified"
fi
if [ "$#" -ne "2" ]; then
    usage
fi

SIZE=`humanreadable $1`
FILENAME="$2"

BS=`humanreadable $bs`

COUNT=`expr $SIZE / $BS`
CHECK=`expr $COUNT \* $BS`
if [ "$CHECK" -ne "$SIZE" ]; then
    echo "Warning: Due to the blocksize requested, the file created will be\"
        "`expr $COUNT \* $BS` bytes and not $SIZE bytes"
fi

echo -en "Creating $SIZE byte file $FILENAME...."

dd if="$INFILE" bs=$BS count=$COUNT of="$FILENAME" 2>/dev/null
ddresult=$?
if [ "$quiet" -ne "1" ]; then
    if [ "$ddresult" -eq "0" ]; then
        echo "Finished:"
    else
        echo "An error occurred. dd returned code $ddresult."
    fi
    # We all know that you're going to do this next - let's do it for you:
    ls -l "$FILENAME" && ls -lh "$FILENAME"
fi

exit $ddresult

```

mkfile

\$ **./mkfile -?**

Usage: mkfile [-i infile] [-q] [-b blocksize] size[k|m|g] filename
 Blocksize is 512 bytes by default.

```

$ ./mkfile -i
./mkfile: option requires an argument -- i
Usage: mkfile [ -i infile ] [ -q ] [ -b blocksize ] size[k|m|g] filename
Blocksize is 512 bytes by default.
$ ./mkfile 10k foo
Creating 10240 byte file foo....Finished:
-rw-rw-r-- 1 steve steve 10240 Jan 28 00:31 foo
-rw-rw-r-- 1 steve steve 10K Jan 28 00:31 foo
$

```

对于 `mkfile`，第一次运行使用 `-?` 选项，在代码中按照自己定义的路线执行，显示用法信息后退出。第二次运行显示是非法输入。`-i` 必须带一个文件名或可以读取的设备。`getopts` 显示错误消息 “option requires an argument--i”，但继续执行。当脚本随后检查 `$#` 是否等于 2 时，由于条件不成立，用法信息被显示出来。最后，第三次运行显示 `mkfile` 脚本执行成功。

8.4.1 错误处理

如果不希望 `getopts` 自身显示错误消息，可以设置 `OPTERR=0` (默认值为 1)。不能导出该变量来对已有的 shell 脚本进行修改。一旦开始一个新的 shell 或 shell 脚本，`OPTERR` 将被重置为 1。将 `OPTERR` 赋值为 0 可以显示与任务更相关的信息，这有助于脚本的显示输出更加流畅。脚本最好能报告错误消息 “Input file not specified; please specify the input file when using the `-i` option”。



除了 `OPTERR` 变量之外，还可以将参数定义的第一个字符设置为一个冒号，即使用 `'i:b:q?'` 而非 `'i:b:q'`。

这一小小的修改使得脚本更加流畅。因为选项被置入到 `OPTARG` 变量中，所以可以根据缺少的值来给出自定义的错误消息。这里使用了嵌套的 `case` 语句来处理 `OPTARG` 的新数值。注意，与问号类似，分号必须用反斜线进行转义或放在引号中。这里给出了两种形式，即使用带反斜线的问号与引号中的分号：



可从
wrox.com
下载源代码

```

while getopts 'i:b:q?' argv
do
    case $argv in
        i) INFILE=$OPTARG ;;
        b) bs=$OPTARG ;;
        q) quiet=1 ;;
        \?) usage ;;
        ":")
            case $OPTARG in
                i) echo "Input file not specified."
                    echo "Please specify the input file when using the -i option."
                    echo
                    usage
                    ;;
            esac
        ;;
    esac
done

```

```

        b) echo "Block size not specified."
           echo "Please specify the block size when using the -b option."
           echo
           usage
           ;;
        *) echo "An unexpected parsing error occurred."
           echo
           usage
           ;;
    esac
    exit 2
esac
done

```

mkfile2

```

$ ./mkfile -i
./mkfile: option requires an argument -- i
Usage: mkfile [ -i infile ] [ -q ] [ -b blocksize ] size[k|m|g] filename
Blocksize is 512 bytes by default.
$ ./mkfile2 -i
Input file not specified.
Please specify the input file when using the -i option.

Usage: mkfile [ -i infile ] [ -q ] [ -b blocksize ] size[k|m|g] filename
Blocksize is 512 bytes by default.
$

```

8.4.2 函数中的 getopt

`getopts` 也可以在函数中使用。下面这个函数将温度从摄氏转换为华氏，且接收各种选项来控制其工作方式。因为随着 `getopts` 对参数进行读取，`OPTIND` 变量的值会增加，所以每次调用函数时都必须对其进行重置。`OPTIND` 是 `getopts` 用来跟踪当前索引的计数器。当只需要进行一次参数分析时，`OPTIND` 的增加一般不会造成任何影响。但是由于函数要被多次调用，因此每次都必须重置 `OPTIND`。



可从
wrox.com
下载源代码

```
$ cat temperature.sh
```

```
#!/bin/bash
```

```
convert()
```

```
{
```

```
    # Set defaults
```

```
    quiet=0
```

```
    scale=0
```

```
    error=0
```

```
    source=centigrade
```

```
    # Reset optind between calls to getopt
```

```
OPTIND=1
while getopts 'c:f:s:q?' opt
do
    case "$opt" in
        "c") centigrade=$OPTARG
              source=centigrade ;;
        "f") fahrenheit=$OPTARG
              source=fahrenheit ;;
        "s") scale=$OPTARG ;;
        "q") quiet=1 ;;
        *) echo "Usage: convert [ -c | -f ] temperature [ -s scale | -q ]"
           error=1
           return 0 ;;
    esac
done

if [ "$quiet" -eq "1" ] && [ "$scale" != "0" ]; then
    echo "Error: Quiet and Scale are mutually exclusive."
    echo "Quiet can only return whole numbers between 0 and 255."
    exit 1
fi

case $source in
    centigrade)
        fahrenheit=`echo scale=$scale \; $centigrade \* 9 / 5 + 32 | bc`
        answer="$centigrade degrees Centigrade is $fahrenheit degrees Fahrenheit"
        result=$fahrenheit
        ;;
    fahrenheit)
        centigrade=`echo scale=$scale \; \($fahrenheit - 32\) \* 5 / 9 | bc`
        answer="$fahrenheit degrees Fahrenheit is $centigrade degrees Centigrade"
        result=$centigrade
        ;;
    *)
        echo "An error occurred."
        exit 0
        ;;
esac

if [ "$quiet" -eq "1" ]; then
    if [ "$result" -gt "255" ] || [ "$result" -lt "0" ]; then
        # scale has already been tested for; it must be an integer.
        echo "An error occurred."
        echo "Can't return values outside the range 0-255 when quiet."
        error=1
        return 0
    fi
    return $result
else
    echo $answer
fi
```

```

}

# Main script starts here.

echo "First by return code..."
convert -q -c $1
result=$?
if [ "$error" -eq "0" ]; then
    echo "${1}C is ${result}F."
fi

convert -f $1 -q
result=$?
if [ "$error" -eq "0" ]; then
    echo "${1}F is ${result}C."
fi

echo

echo "Then within the function..."
convert -f $1
convert -c $1

echo

echo "And now with more precision..."
convert -f $1 -s 2
convert -s 3 -c $1

```

temperature.sh

```

$ ./temperature.sh 12
First by return code...
12C is 53F.
An error occurred.
Can't return values outside the range 0-255 when quiet.

Then within the function...
12 degrees Fahrenheit is -11 degrees Centigrade
12 degrees Centigrade is 53 degrees Fahrenheit

And now with more precision...
12 degrees Fahrenheit is -11.11 degrees Centigrade
12 degrees Centigrade is 53.600 degrees Fahrenheit

```



第二次测试失败，因为答案(12F=-11C)不能表示为函数的返回码(包含1个字节，范围只能是0~255)。奇怪的是，函数不总是最适于用来返回数值输出的方法。

8.5 本章小结

我们可以使用函数将代码块打包成灵活、模块化与可重用的形式，并且使用方便。函数的调用方式与 shell 脚本以及其他命令的调用方式非常相似，并且参数分析方式与 shell 脚本完全相同——\$1、\$2、\$3 与 \$*，另外 getopt 在函数与 shell 脚本中的工作方式都一样。因此可以非常容易地将一个已有脚本修改成一个函数，反之亦然。

将一些相关的函数组织在一起形成库可以对语言进行扩展，并且能针对手头的特定任务提供自定义的命令。库可以实现细节与脚本分离并将这些细节保存在更适当的位置，使得编程更容易、更抽象并且更方便。这使得库的工作方式变得透明，使得库中的方法能实现无缝修改与升级。

尽管数组不能很好地适用于函数，但它仍是一类非常灵活的特殊变量。第 9 章将详细介绍数组。

数 组

数组是一种包含值的集合的特殊变量，可以通过键(也可以称为索引)来访问它。除非另外指定，否则 `bash` 中的数组都从 0 开始索引，所以数组的第一个元素是 `${array[0]}` 而不是 `${array[1]}`。并不是每个人都觉得这样比较直观，但它体现了一个事实，那就是 `bash` 是用 C 语言编写的，而 C 也是从 0 开始索引数组。

可以创建稀疏数组。所以对于像 PID 到进程名称的映射这样的非连续数据，我们可以存储 `pid[35420]=httpd -k ssl`，并且不需要将其他 35 419 项全部存储在数组中。尽管很难得知哪些索引实际存储了值，但这还是非常有用。

`shell` 中的数组只能是一维的。也就是说，如果要对一个棋盘建模，则不能用 `${chessboard[2][5]}` 来访问 c6 方格。然而，我们必须找到一种方法，它可以将棋盘平面化到一个线性的包含 64 个元素的数组上。所以 `${chessboard[0]}` 到 `${chessboard[7]}` 是第一行，`${chessboard[8]}` 到 `${chessboard[15]}` 是第二行，依此类推。另一种方法是使用 8 个 8 元素的数组。本书后面的实用脚本 17-1 就使用这种方法来处理多行字符。

`bash 4.0` 新增了关联数组。这些数组将文本而不是数字作为其索引，所以可以使用 `${points[Ferrari]}` 与 `${points[McLaren]}` 而不是 `${points[0]}` 与 `${points[1]}` 来跟踪比赛结果，然后再用一个查找表将 0 与 1 分别映射到 Ferrari 与 McLaren。本章介绍各种可供使用的不同类型的数组，以及它们的用途与访问、操作方法。

9.1 数组的赋值

有好几种不同的为数组赋值的方式。尽管有些比较类似，但有 3 种主要的赋值方式，分别为“一次一个”、“一次全部”或者介于前两者之间的“按索引”。还可以从各种源来对数组赋值，包括通配符扩展与程序输出。

如果数组是通过“一次一个”或者“一次全部”的方法声明的，那么 `shell` 自动检测是否有数组正在被声明。另外，语句 `declare -a myarray` 可以用来向 `shell` 声明该变量是被作为数组来使用的。

9.1.1 一次一个

对数组赋值的最简单直接的方式是每次对一个元素进行赋值。与常规变量一样，赋值时不使用美元符号(\$)，只在引用时才使用。变量名后面的索引号要使用方括号括起来：

```
numberarray[0]=zero
numberarray[1]=one
numberarray[2]=two
numberarray[3]=three
```

除了简洁与清晰，这种方式的另一个优势是可以定义稀疏数组。下例中的数组就没有第3项(two)：

```
numberarray[0]=zero
numberarray[1]=one
numberarray[3]=three
```

与常规变量一样，值可以包含空格，但需要用引号或反斜线将其引用起来。

```
country[0]="United States of America"
country[1]=United\ Kingdom
country[2]=Canada
country[3]=Australia
```

9.1.2 一次全部

更高效的给数组赋值的方式是在单个命令中列出所有的值。方法是在括号中列出所有用空格分开的值：

```
students=( Dave Jennifer Michael Alistair Lucy Richard Elizabeth )
```

缺点是稀疏数组不能这样赋值。另一个缺点是必须预先知道要赋的值，并且能够将值硬编码到脚本中或者由脚本自行计算其值。

IFS 中的任何字符都能用来将元素分隔开，包括换行符。将数组的定义分散在多行是完全合法的。我们甚至可以使用注释来结束一行。在下面的代码中，`students` 被分割成单独的几行，每行以一个注释结束，每行的列表子集表示每组学生代表的那个年份。



IFS——内部字段分隔符(Internal Field Separator)——在第3章中有介绍。



可从

wrox.com
下载源代码

```
$ cat studentarray.sh
#!/bin/bash
```

```
students=( Dave Jennifer Michael      # year 1
```

```

Alistair Lucy Richard Elizabeth    # year 2
Albert Roger Dennis James Roy      # year 3
Rory Jim Andi Elaine Clive        # year 4
)

for name in ${students[@]}
do
    echo -en "$name "
done
echo
$ ./studentarray.sh
Dave Jennifer Michael Alistair Lucy Richard Elizabeth Albert Roger Dennis James Roy
Rory Jim Andi Elaine Clive
$

```

studentarray.sh

实际上，对于一些非一次性任务，将学生姓名硬编码到脚本中并不太现实。本书后面将介绍更灵活的数据读取方法。

9.1.3 按索引

这种方法是“一次一个”的简易版本，或者也可以把它看成是“一次全部”的更显式的方法。这种方法在一对括号中对值一起进行赋值，但索引与值是相对应的。这种方法主要用于创建稀疏数组，但也能用于清晰地描述元素的索引位置，而不用冗长的“一次一个”方法，因为它每次都需要给出变量名。这种“按索引”的方法在使用本章后面将介绍的关联数组时也特别有用。下面的代码在一个数组中为前32个ASCII字符赋予了名称(参见 `man ascii`)。这可以用于确保名称处于正确的位置。例如，我们可以不用从开始数到13便很容易判断出CR位于索引13处。

```

nonprinting=( [0]=NUL [1]=SOH [2]=STX [3]=ETX [4]=EOT [5]=ENQ
[6]=ACK [7]=BEL [8]=BS [9]=HT [10]=LF [11]=VT [12]=FF [13]=CR
[14]=SO [15]=SI [16]=DLE [17]=DC1 [18]=DC2 [19]=DC3 [20]=DC4
[21]=NAK [22]=SYN [23]=ETB [24]=CAN [25]=EM [26]=SUB [27]=ESC
[28]=FS [29]=GS [30]=RS [31]=US)

```

9.1.4 从源中一次全部读取

这种方法是“一次全部”的特殊情况——括号中的内容可以由shell自身来提供，无论来自文件名扩展还是任何命令或函数的输出。要读取进程状态表中的数值，将每个值都赋给数组的元素，我们只需要调用(`$(cat /proc/$$/stat)`)。输出的每一项都会赋给数组的一个元素。

```

$ cat /proc/$$/stat
28510 (bash) S 28509 28510 28510 34818 28680 4202496 3094 49631 1 1 6 14 123 28 20
0 1 0 27764756 20000768 594 18446744073709551615 4194304 5082140 140736253670848
140736253669792 140176010326894 0 69632 3686404 1266761467 0 0 0 17 0 0 0 92 0 0

```

```
$ stat=( $(cat /proc/$$/stat) )
$ echo ${stat[1]}
(bash)
$ echo ${stat[2]}
S
$ echo ${stat[34]}
0
$ echo ${stat[23]}
594
$
```

如果要逐行读取文件，则将 IFS 设置为换行符后再读取。这一技术在将文本文件读取到内存中时特别有用。



```
$ cat readhosts.sh
```

```
#!/bin/bash
```

```
oIFS=$IFS
```

```
IFS="
```

```
"
```

```
hosts=( `cat /etc/hosts` )
```

```
for hostline in "${hosts[@]}"
```

```
do
```

```
    echo line: $hostline
```

```
done
```

```
# always restore IFS or insanity will follow...
```

```
IFS=$oIFS
```

```
$ ./readhosts.sh
```

```
line: 127.0.0.1 localhost
```

```
line: # The following lines are desirable for IPv6 capable hosts
```

```
line: ::1 localhost ip6-localhost ip6-loopback
```

```
line: fe00::0 ip6-localnet
```

```
line: ff00::0 ip6-mcastprefix
```

```
line: ff02::1 ip6-allnodes
```

```
line: ff02::2 ip6-allrouters
```

```
line: 192.168.1.3      sky
```

```
line: 192.168.1.5      plug
```

```
line: 192.168.1.10     declan
```

```
line: 192.168.1.11     atomic
```

```
line: 192.168.1.12     jackie
```

```
line: 192.168.1.13     goldie smf
```

```
line: 192.168.1.227    elvis
```

```
line: 192.168.0.210    dgoldie ksgp
```

```
$
```

Readhosts.sh

读取的源可以是通配符扩展得到的一系列文件。下面的代码中，与模式*.mp3 匹配的

所有文件都被添加到 mp3s 数组中:

```
$ mp3s=( *.mp3 )
$ for mp3 in "${mp3s[@]}"
> do
>   echo "MP3 File: $mp3"
> done
MP3 File: 01 - The MC5 - Kick Out The Jams.mp3
MP3 File: 02 - Velvet Underground - I'm Waiting For The Man.mp3
MP3 File: 03 - The Stooges - No Fun.mp3
MP3 File: 04 - The Doors - L.A. Woman.mp3
MP3 File: 05 - The New York Dolls - Jet Boy.mp3
MP3 File: 06 - Patti Smith - Gloria.mp3
MP3 File: 07 - The Damned - Neat Neat Neat.mp3
MP3 File: 08 - X-Ray Spex - Oh Bondage Up Yours!.mp3
MP3 File: 09 - Richard Hell & The Voidoids - Blank Generation.mp3
MP3 File: 10 - Dead Boys - Sonic Reducer.mp3
MP3 File: 11 - Iggy Pop - Lust For Life.mp3
MP3 File: 12 - The Saints - This Perfect Day.mp3
MP3 File: 13 - Ramones - Sheena Is A Punk Rocker.mp3
MP3 File: 14 - The Only Ones - Another Girl, Another Planet.mp3
MP3 File: 15 - Siouxsie & The Banshees - Hong Kong Garden.mp3
MP3 File: 16 - Blondie - One Way Or Another.mp3
MP3 File: 17 - Magazine - Shot By Both Sides.mp3
MP3 File: 18 - Buzzcocks - Ever Fallen In Love (With Someone You
Shouldn't've).mp3
MP3 File: 19 - XTC - This Is Pop.mp3
MP3 File: 20 - Television - Marquee Moon.mp3
MP3 File: 21 - David Bowie - 'Heroes'.mp3
$
```



注意这些文件名中所有存在潜在问题的字符。其中有空格、逗号、&符号、单引号、括号与点号。在这个简单的例子中，数组结构可以很容易地处理这些字符。本章后面将对这些字符串进行更复杂的操作，并且程序员无须担心任何特殊的引用。

9.1.5 从输入读取

使用 -a 标志调用 `bash shell` 的内置命令 `read` 可以将元素读取到数组中。无论是从文件或是用户输入，这都是定义数组的简单方法。

```
$ read -a dice
4 2 6
$ echo "you rolled ${dice[0]} then ${dice[1]} then ${dice[2]}"
you rolled 4 then 2 then 6
$
```

`read -a` 命令在很多场景下都特别有用，而与 `IFS` 变量组合时则更加威力无穷。下面的例子告诉 `shell`，`IFS` 不同于默认值 `<space><tab><newline>`，所以要读取字段由分号隔开的 `/etc/passwd`，只需要在 `read` 命令之前设置 `IFS=:`。注意，只要可能，最好在需要修改 `IFS` 的命令的同一行上使用 `IFS=` 语句，因为这样只会对 `read` 命令有效，而不会影响到循环中的其他命令。`GECOS` 字段可能包含一些尾随的逗号。通常情况下应该是 `Steve Parker,The Lair,202-456-1414,Author`，但本例中只是 `Steve Parker,,,`。这样看起来不美观，所以使用 `%%,*` 语法将这些尾随逗号去掉。一些系统账户根本没有 `GECOS` 字段，所以如果没有设置第 5 个字段(`GECOS`)，则该脚本退而用第 1 个字段(登录名)来定义 `$user` 变量。



可从
wrox.com
下载源代码

```
$ cat user.sh
```

```
#!/bin/bash
```

```
while IFS=: read -a userdetails
do
```

```
    unset user
```

```
    gecos=${userdetails[4]%%,*}
```

```
    username=${userdetails[0]}
```

```
    user=${gecos:-$username}
```

```
    if [ -d "${userdetails[5]}" ]; then
```

```
        echo "${user}'s directory ${userdetails[5]} exists"
```

```
    else
```

```
        echo "${user}'s directory ${userdetails[5]} doesn't exist"
```

```
    fi
```

```
done < /etc/passwd
```

```
$ ./user.sh
```

```
root's directory /root exists
```

```
daemon's directory /usr/sbin exists
```

```
bin's directory /bin exists
```

```
sys's directory /dev exists
```

```
sync's directory /bin exists
```

```
games's directory /usr/games exists
```

```
man's directory /var/cache/man exists
```

```
lp's directory /var/spool/lpd doesn't exist
```

```
mail's directory /var/mail exists
```

```
news's directory /var/spool/news doesn't exist
```

```
uucp's directory /var/spool/uucp doesn't exist
```

```
proxy's directory /bin exists
```

```
www-data's directory /var/www exists
```

```
backup's directory /var/backups exists
```

```
Mailing List Manager's directory /var/list doesn't exist
```

```
ircd's directory /var/run/ircd doesn't exist
```

```
Gnats Bug-Reporting System (admin)'s directory /var/lib/gnats doesn't exist
```

```
nobody's directory /nonexistent doesn't exist
```

```
libuuid's directory /var/lib/libuuid exists
```

```
messagebus's directory /var/run/dbus exists
```

```
Avahi autoip daemon's directory /var/lib/avahi-autoipd exists
```

```
festival's directory /home/festival doesn't exist
```

```
Gnome Display Manager's directory /var/lib/gdm exists
```

```

Hardware abstraction layer's directory /var/run/hald exists
usbmux daemon's directory /home/usbmux doesn't exist
sshd's directory /var/run/sshd exists
saned's directory /home/saned doesn't exist
HPLIP system user's directory /var/run/hplip exists
Steve Parker's directory /home/steve exists
Avahi mDNS daemon's directory /var/run/avahi-daemon exists
ntp's directory /home/ntp doesn't exist
Debian-exim's directory /var/spool/exim4 exists
TiMidity++ MIDI sequencer service's directory /etc/timidity exists
$

```

user.sh

bash shell 的内置命令 **readarray** 能够用比之前读取 `/etc/hosts` 的脚本更加灵活的方式来读取文本文件。可以指定初始的索引值(-O)，还可以指定读取的最大行数(-n)。另外还可以从输入的头跳过几行(-s)。

```

$ readarray -n 4 -s 2 food
porridge
black pudding
apples
bananas
cucumbers
burgers
eggs
$ printf "%s" "${food[@]}"
apples
bananas
cucumbers
burgers
$

```

上面的命令使用参数 **-s 2** 跳过了前两项。尽管这意味着一共读取了 6 个参数，但因为参数 **-n 4**，实际读取的参数只有 4 个，所以数组的大小为 4。输入的第 7 项完全不会被读取到。

通常用来显示数组元素的方法是 `printf "%s\n" "${food[@]}"`。该语句遍历数组的所有值，并在每个显示出来的字符串后面添加一个换行符。因为结尾的换行符被添加到数组元素中，所以上面的例子不再需要 `\n`。**readarray** 的 **-t** 标志可以将几乎总是需要的结尾换行符去掉。

9.2 数组的访问

访问数组值的基本方法与之前介绍的数组赋值方法非常相似。访问必须使用花括号。如果省略索引，则假设访问第一个元素。

9.2.1 用索引访问

下面的代码重用了本章前面用到的 **numberarray** 数组，并使用一些 **echo** 语句来显示赋

值之后的值。注意，稀疏数组意味着不存在元素 `numberarray[2]`。

```
$ numberarray[0]=zero
$ numberarray[1]=one
$ numberarray[3]=three
$ echo ${numberarray[0]}
zero
$ echo ${numberarray[2]}

$ echo ${numberarray[3]}
three
$
```

如果不用花括号来访问 `$numberarray[1]`，shell 将会把 `$numberarray` 解释成 `numberarray` 中的第一个元素，并且 `[1]` 是字符串字面量。这会返回字符串字面量 `zero[1]`，但不是我们想要的结果。

```
$ echo $numberarray[1]
zero[1]
$
```

9.2.2 数组的长度

计算数组元素的数量与计算常规变量的长度非常相似。`${#myvar}` 返回 `$myvar` 变量中字符串的长度，`${#myarray[@]}` 或者 `${#myarray[*]}` 返回数组中的元素个数。对于稀疏数组，返回的还只是数组中实际赋过值的元素数目，这与数组使用的最大索引不相同。

同样要注意的是，`${#myarray}` 返回的是 `${myarray[0]}` 中字符串的长度，而不是数组 `$myarray` 的元素数目。要获取数组中第三个元素的长度，需要使用语法 `${#array[2]}`，因为数组是从 0 开始索引的。`${#array[0]}` 是第一个元素的长度，`${#array[1]}` 是第二个元素的长度。

```
$ fruits=( apple banana pear orange )
$ echo ${#fruits[@]}
4
$ echo ${#fruits}
5
$ echo ${#fruits[0]}
5
$ echo ${#fruits[1]}
6
$ echo ${#fruits[2]}
4
$ echo ${#fruits[3]}
6
$
```

9.2.3 用变量索引访问

索引不必是硬编码出来的整数，还可以是其他变量值。所以，我们可以使用变量来遍历一个(非稀疏)数组，甚至可以使用索引来随机访问数组中的元素。下面的例子显示了如何遍历 4 个披头士，他们的索引号是 0~3。命令 `seq 0 $((${#beatles[@]} - 1))` 从 0 数到 3，或者准确来说是从 0 数到 $(4 - 1)$ 。这里的 4 是指数组的长度，但因为数组是从 0 开始索引的，所以 4 个元素的索引是从 0 到 3。

该脚本随后添加了第 4 个元素，索引为 5，数组成为了一个稀疏数组(没有元素 4)，所以 Stuart Sutcliffe(或者是 Pete Best)不能使用这个循环来访问。



可从
wrox.com
下载源代码

```
$ cat index.sh
#!/bin/bash

beatles=( John Paul Ringo George )
for index in $(seq 0 $(( ${#beatles[@]} - 1 )))
do
    echo "Beatle $index is ${beatles[$index]}."
done

echo "Now again with the fifth beatle..."
beatles[5]=Stuart

for index in $(seq 0 $(( ${#beatles[@]} - 1 )))
do
    echo "Beatle $index is ${beatles[$index]}."
done
echo "Missed it; Beatle 5 is ${beatles[5]}."
$ ./index.sh
Beatle 0 is John.
Beatle 1 is Paul.
Beatle 2 is Ringo.
Beatle 3 is George.
Now again with the fifth beatle...
Beatle 0 is John.
Beatle 1 is Paul.
Beatle 2 is Ringo.
Beatle 3 is George.
Beatle 4 is .
Missed it; Beatle 5 is Stuart.
$
```

index.sh

只要数组不是关联型的，我们就还可以在方括号中执行基本的数学操作。这不仅可以使代码容易编写，并且能保持很好的可读性。冒泡算法将元素与其相邻元素进行比较，而比起必须包含计算 $j - 1$ 的额外代码来，下面的代码更容易读懂。使用 C 风格的 for 循环让下面的代码不太像是 shell 脚本。



可从
wrox.com
下载源代码

```
$ cat bubblesort.sh
```

```
#!/bin/bash
```

```
function bubblesort()
```

```
{
```

```
    n=${#data[@]}
```

```
    for i in `seq 0 $n`
```

```
    do
```

```
        for (( j=n; j > i; j-=1 ))
```

```
        do
```

```
            if [[ ${data[j-1]} > ${data[j]} ]]
```

```
            then
```

```
                temp=${data[j]}
```

```
                data[j]=${data[j-1]}
```

```
                data[j-1]=$temp
```

```
            fi
```

```
        done
```

```
    done
```

```
}
```

```
data=( roger oscar charlie kilo indigo tango )
```

```
echo "Initial state:"
```

```
for i in ${data[@]}
```

```
do
```

```
    echo "$i"
```

```
done
```

```
bubblesort
```

```
echo
```

```
echo "Final state:"
```

```
for i in ${data[@]}
```

```
do
```

```
    echo "$i"
```

```
done
```

```
$ ./bubblesort.sh
```

```
Initial state:
```

```
roger
```

```
oscar
```

```
charlie
```

```
kilo
```

```
indigo
```

```
tango
```

```
Final state:
```

```
charlie
```

```
indigo
```

```
kilo
oscar
roger
tango
$
```

bubblesort.sh



这种形式的缺点是变量\${n}必须在 for 循环之外计算。

9.2.4 从数组中选择元素

从数组中选择单个元素很简单，但有时需要从其中检索一定范围内的元素。这可以通过类似 `substr` 的方式来实现，方法是给数组提供起始索引与要获取的元素数目。简单的 `${food[@]:0:1}` 可以得到第一个元素，`:1:1` 得到第二个，`:2:1` 得到第三个，依此类推。这与使用 `${food[0]}`、`${food[1]}` 与 `${food[2]}` 是一样的，而且作用不是很明显。



如下面例子中所示，访问数组中不存在的第 8 个元素(`${food[@]:7:1}`)不会导致任何错误，只是返回一个空白字符串。

```
$ food=( apples bananas cucumbers dates eggs fajitas grapes )
$ echo ${food[@]:0:1}
apples
$ echo ${food[@]:1:1}
bananas
$ echo ${food[@]:2:1}
cucumbers
$ echo ${food[@]:7:1}

$
```

对获取元素数目进行扩展可以得到更加灵活的机制。将之前代码中最后的`:1` 替换成其他数可以从数组中获取一组结果。

```
$ echo ${food[@]:2:4}
cucumbers dates eggs fajitas
$ echo ${food[@]:0:3}
apples bananas cucumbers
$
```

如果更进一步，我们可以将初始值省略。这样得到的是从为其提供的某个偏移量开始往后的所有元素。

```
$ echo ${food[@]:3}
dates eggs fajitas grapes
$ echo ${food[@]:1}
bananas cucumbers dates eggs fajitas grapes
$ echo ${food[@]:6}
grapes
$
```

9.2.5 显示整个数组

只要一条简单的 `echo ${array[@]}` 命令就能显示整个变量，但这没有太大的吸引力：

```
$ echo ${distros[@]}
Ubuntu Fedora Debian openSuSE Sabayon Arch Puppy
```

更灵活的方式是使用 `printf` 添加文本并格式化到输出。`printf` 将使用同一个格式化字符串来格式化数组中的每个元素，所以它几乎是对一个循环的模拟：

```
$ printf "Distro: %s\n" "${distros[@]}"
Distro: Ubuntu
Distro: Fedora
Distro: Debian
Distro: openSuSE
Distro: Sabayon
Distro: Arch
Distro: Puppy
$
```

9.3 关联数组

关联数组是 `bash` 4.0 新增的一个特性。关联数组将值与索引连接(关联)到一起，所以我们可以将元数据与实际数据关联起来。使用这种方式可以将音乐家与他的乐器联系起来。关联数组必须以大写的 `declare -A` 命令来进行声明。



可从
wrox.com
下载源代码

```
$ cat musicians.sh
#!/bin/bash

declare -A beatles
beatles=( [singer]=John [bassist]=Paul [drummer]=Ringo [guitarist]=George )

for musician in singer bassist drummer guitarist
do
    echo "The ${musician} is ${beatles[$musician]}."
done
$ ./musicians.sh
The singer is John.
```

```
The bassist is Paul.
The drummer is Ringo.
The guitarist is George.
$
```

musicians.sh



在 bash 引入关联数组之前, `${beatles[index]}` 没有任何意义。它会被自动解释为 `${beatles[$index]}`。然而, 因为关联数组可以将文本作为其索引, 所以现在使用 `${beatles["index"]}` 是合法的。因此关联数组中必须使用美元符号。如果省略美元符号, 解释为数组索引的将会是单词 `index`, 而不是变量 `$index` 的值。

关联数组更有用的功能是对索引的名称进行反向引用。这意味着如果给定了乐器的名称, 我们可以得到音乐家的名字。同样地, 给定了音乐家的名字, 我们可以确定乐器。反向引用的语法是 `${!array[@]}`。



可从
wrox.com
下载源代码

```
$ cat instruments.sh
#!/bin/bash

declare -A beatles
beatles=( [singer]=John [bassist]=Paul [drummer]=Ringo [guitarist]=George )
for instrument in ${!beatles[@]}
do
    echo "The ${instrument} is ${beatles[$instrument]}"
done
$ ./instruments.sh
The singer is John
The guitarist is George
The bassist is Paul
The drummer is Ringo
$
```

instruments.sh

9.4 数组操作

数组与其他变量在结构上是有差异的。这意味着对数组进行操作需要一些新的语法。只要可能, 对数组的操作在语法上与对字符串做同样的操作都非常相似。但有些情况下的语法不够灵活。

9.4.1 数组的复制

将数组复制到另一个数组非常简单。对于引用与空格而言, 重要的是使用格式 `${array[@]}` (而不是 `${array[*]}`), 并且要用双引号将整个结构括起来。下面的例子非常清晰地显示了使用其他形式时产生的效果。

```
$ activities=( swimming "water skiing" canoeing "white-water rafting" surfing )
$ for act in ${activities[@]}
> do
>   echo "Activity: $act"
> done
Activity: swimming
Activity: water
Activity: skiing
Activity: canoeing
Activity: white-water
Activity: rafting
Activity: surfing
$
```

导致上面结果的原因在于列表两旁没有使用双引号，所以 swimming、water 与 skiing 都被当成独立的单词。在周围添加双引号可以修正这一问题：

```
$ for act in "${activities[@]}"
> do
>   echo "Activity: $act"
> done
Activity: swimming
Activity: water skiing
Activity: canoeing
Activity: white-water rafting
Activity: surfing
$
```

同样地，星号*对于引用与不引用都不合适。如果没有引用，它与@符号作用相同。如果使用引用，整个数组会被归结为单个字符串。

```
$ for act in ${activities[*]}
> do
>   echo "Activity: $act"
> done
Activity: swimming
Activity: water
Activity: skiing
Activity: canoeing
Activity: white-water
Activity: rafting
Activity: surfing
$ for act in "${activities[*]}"
> do
>   echo "Activity: $act"
> done
Activity: swimming water skiing canoeing white-water rafting surfing
$
```

因此, 可以用"`${activities[@]}`"的值定义一个新的数组来对原来的数组进行复制。这样会按照相同的方式保留空白字符, 正如上面代码中的 `for` 循环显示的一样, 空白字符被正确地保留下来。代码如下:

```
$ hobbies=( "${activities[@]}" )
$ for hobby in "${hobbies[@]}"
> do
>   echo "Hobby: $hobby"
> done
Hobby: swimming
Hobby: water skiing
Hobby: canoeing
Hobby: white-water rafting
Hobby: surfing
$
```

然而, 这种方式不适用于稀疏数组。索引的实际值不是按照这种方式来传递的, 所以 `hobbies` 数组不是 `activities` 数组的真实副本。

```
$ activities[10]="scuba diving"
$ hobbies="( ${activities[@]} )"
$ for act in `seq 0 10`
> do
>   echo "$act : ${activities[$act]} / ${hobbies[$act]}"
> done
0 : swimming / swimming
1 : water skiing / water skiing
2 : canoeing / canoeing
3 : white-water rafting / white-water rafting
4 : surfing / surfing
5 : / scuba diving
6 : /
7 : /
8 : /
9 : /
10 : scuba diving /
$
```

9.4.2 向数组追加元素

向数组追加元素的方法与数组复制非常类似。最简单的追加到数组的方法是将数组复制语句进行扩展。

```
$ hobbies=( "${activities[@]}" diving )
$ for hobby in "${hobbies[@]}"
> do
>   echo "Hobby: $hobby"
> done
```

```
Hobby: swimming
Hobby: water skiing
Hobby: canoeing
Hobby: white-water rafting
Hobby: surfing
Hobby: scuba diving
Hobby: diving
$
```

本章开头介绍了如何使用 `seq 0 ${#beatles[@]} - 1` 获取数组的最后一个实际元素。但数组从 0 开始索引这一事实使得这一任务变得有些棘手。在向数组追加单个元素时，数组从 0 开始索引实际上使得追加操作更容易。

```
$ hobbies[${#hobbies[@]}]=rowing
$ for hobby in "${hobbies[@]}"
> do
>   echo "Hobby: $hobby"
> done
Hobby: swimming
Hobby: water skiing
Hobby: canoeing
Hobby: white-water rafting
Hobby: surfing
Hobby: scuba diving
Hobby: diving
Hobby: rowing
$
```

bash shell 确实有组合两个数组的内置语法。这种使用 C 风格符号 `+=` 的方法更简洁，而且写出的代码更清晰。

```
$ airports=( flying gliding parachuting )
$ activities+=("${airports[@]}")
$ for act in "${activities[@]}"
> do
>   echo "Activity: $act"
> done
Activity: swimming
Activity: water skiing
Activity: canoeing
Activity: white-water rafting
Activity: surfing
Activity: scuba diving
Activity: climbing
Activity: walking
Activity: cycling
Activity: flying
Activity: gliding
Activity: parachuting
$
```

9.4.3 从数组中删除元素

从数组中删除元素与删除变量相同。我们可以使用 `myarray[3]=` 或者 `unset myarray[3]`。同样还可以 `unset` 整个数组。然而，`myarray=` 本身只会清除数组中第一个元素的值。所有这些情况都出现在下面的代码中。

```
$ for act in `seq 0 ${#activities[@]} - 1`)`
> do
>   echo "Activity $act: ${activities[$act]}"
> done
Activity 0: swimming
Activity 1: water skiing
Activity 2: canoeing
Activity 3: white-water rafting
Activity 4: surfing
Activity 5: scuba diving
Activity 6: climbing
Activity 7: walking
Activity 8: cycling
Activity 9: flying
Activity 10: gliding
Activity 11: parachuting
$ activities[7]=
$ for act in `seq 0 ${#activities[@]} - 1`)`
> do
>   echo "Activity $act: ${activities[$act]}"
> done
Activity 0: swimming
Activity 1: water skiing
Activity 2: canoeing
Activity 3: white-water rafting
Activity 4: surfing
Activity 5: scuba diving
Activity 6: climbing
Activity 7:
Activity 8: cycling
Activity 9: flying
Activity 10: gliding
Activity 11: parachuting
$
```

这样做的结果是得到一个稀疏数组。使用 `unset activities[7]` 几乎可以得到相同的效果。如第7章讨论过的，将变量赋值为空字符串与完全删除是有区别的，但它们的区别只有在 `使用 ${variable+string}` 或 `使用 ${variable?string}` 形式时才会比较明显。

```
$ echo ${activities[7]}

$ echo ${activities[7]+"Item 7 is set"}
Item 7 is set
```



```
$ unset activities[7]
$ echo ${activities[7]}+"Item 7 is set"

$
```

另外，不使用索引来引用数组会被解释成对数组中第一个元素的引用。因此，按照这样的方式清除数组只会删除数组的第一项。

```
$ activities=
$ for act in `seq 0 ${#activities[@]} - 1)`
> do
>   echo "Activity $act: ${activities[$act]}"
> done
Activity 0:
Activity 1: water skiing
Activity 2: canoeing
Activity 3: white-water rafting
Activity 4: surfing
Activity 5: scuba diving
Activity 6: climbing
Activity 7:
Activity 8: cycling
Activity 9: flying
Activity 10: gliding
Activity 11: parachuting
```

如果对 `activities` 数组本身进行 `unset`，则这个数组都会消失。尽管可以使用 `unset myarray[*]`，但将数组整个删除才是正确的。

```
$ unset activities
$ for act in `seq 0 ${#activities[@]} - 1)`
> do
>   echo "Activity $act: ${activities[$act]}"
> done
$
```

9.5 高级技术

在 9.1.4 节中，我们使用一个清晰的模式将一个 MP3 文件集合列了出来。虽然这一模式几乎也能用于排序，但并不十分理想。其中将艺术家与音轨号分开的连字符以并不十分完美的方式自动对 MP3 文件进行标记，但是用 `cut` 或 `awk` 命令来对它们进行剪裁就会有些棘手。冒号能比空格-短划线-空格更好地对字段进行分离。没有最理想的字符，因为冒号可能出现在歌曲名或艺术家名字中，但这一简短的脚本将较复杂的文件名列表缩减到基本形式。

尽管是对整个数组进行操作，但这次 `for` 语句本身从文件名中去掉了结尾的 `.mp3`，并

且与第7章介绍的方法一样。之后，当 `echo` 访问 `${mp3}` 变量时，也同时用冒号代替了短划线(-)。这并不是说代码运行起来会比分别完成这些任务要更快，但这确实是一种清晰而简单的代码编写方法。通常而言最重要的是代码要容易理解。

```
$ for mp3 in "${mp3s[@]}/%.mp3"
> do
>   echo ${mp3// - /:}
> done
01:The MC5:Kick Out The Jams
02:Velvet Underground:I'm Waiting For The Man
03:The Stooges:No Fun
04:The Doors:L.A. Woman
05:The New York Dolls:Jet Boy
06:Patti Smith:Gloria
07:The Damned:Neat Neat Neat
08:X-Ray Spex:Oh Bondage Up Yours!
09:Richard Hell & The Voidoids:Blank Generation
10:Dead Boys:Sonic Reducer
11:Iggy Pop:Lust For Life
12:The Saints:This Perfect Day
13:Ramones:Sheena Is A Punk Rocker
14:The Only Ones:Another Girl, Another Planet
15:Siouxsie & The Banshees:Hong Kong Garden
16:Blondie:One Way Or Another
17:Magazine:Shot By Both Sides
18:Buzzcocks:Ever Fallen In Love (With Someone You Shouldn't've)
19:XTC:This Is Pop
20:Television:Marquee Moon
21:David Bowie:'Heroes'
$
```

9.6 本章小结

数组是 `shell` 的强大特性，并且用法多样。如果需要使用 `shell` 的数组特性，则会限制 `shell` 脚本向不支持数组的系统的可移植性。尽管如此，数组还是极大地扩展了 `shell` 的功能。稀疏数组也非常有用，并且不用像 C 语言这样的低级语言一样要对内存进行管理。

关联数组比常规数组甚至更加灵活。实际上，它们与将名称而不是数字作为索引号的稀疏数组比较类似。这允许我们在数据的键上存储元数据。我们可以将数组中的每个元素当成独立的字符串，并对这些字符串使用更加高级的 `bash` 特性。这样可以获得极大的灵活性，否则必须借助 `perl`、`awk` 或 `sed`。

第10章将介绍进程以及内核对进程管理的方式。内容涉及前台与后台进程、信号、`exec`，以及 I/O 重定向与管道。内核负责控制系统中的所有进程，以及它们的运行方式与相互之间的交互方式。Linux 内核还通过 `/proc` 虚拟文件系统来显示内核中大量的内部信息，这对于要访问内核内部数据结构的 `shell` 脚本非常有用。

第 10 章

进 程

操作系统的一个主要任务就是从用户、服务与应用程序的角度运行进程。这些进程由内核中的一个进程表进行跟踪。进程表包含了系统中每个进程的当前状态，并且系统调度器决定每个进程何时分配到一个 CPU，以及何时与 CPU 分离。`ps` 命令可以查询进程表。按照“做一件事并把它做好”的 Unix 模型，`ps` 有一组开关用来调整从整个进程树中显示哪些信息。

利用 `/proc` 虚拟文件系统能进一步窥视到正在运行的内核。进程表中的每个进程在 `/proc` 下都有一个目录。从进程的目录中可以找到进程的状态——它的当前目录与当前打开的文件。Linux 内核能给出比进程表多得多的操作系统信息。`/proc` 包含了直接读写内核状态的机制，包括网络设置、内存选项、硬件信息，甚至可以强制机器崩溃。

本章将介绍进程的概念、进程的管理方式与操作方法。进程是 Unix 最古老的概念之一，并且 40 年来没有太大变化。但 Linux 内核特别引入了一种新的关联方式，在 `/proc` 文件系统中为内核与用户空间提供真正双向交互。

10.1 `ps` 命令

`ps` 命令在不同的 Unix 下不太一致。System V Unix 使用普通的短划线(-)来表示选项，所以 `ps -eaf` 只是一条普通的显示所有当前进程的命令；而 BSD Unix 则不同，并且还能识别不同的开关。`ps aux` 是 BSD 系统中显示当前所有进程的常规方式。通常情况下，GNU/Linux 会涵盖这两种风格。`ps` 的 GNU 实现接受 System V 与 BSD 选项，以及自身的 GNU 选项，如 `--user`。System V 格式可能使用最广泛。本章将使用 System V 风格。

```
$ ps -fp 3010
```

```
UID      PID PPID C STIME TTY      TIME CMD
Mysql    3010 2973 0 Oct24 ?    00:11:23 /usr/sbin/mysqld --basedir=/usr --d
atadir=/var/lib/mysql --user=mysql --pid-file=/var/run/mysqld/mysqld.pid
--skip-external-locking --port=3306
```

标题的含义有些模糊。UID、PID 与 PPID 分别是用户名、进程 ID、父进程 ID。STIME

是进程开始的时间(或日期)。如果与某个终端关联起来,那么会在 TTY 下报告终端。TIME 是进程使用的 CPU 时间, CMD 是可执行文件的全称。C 是对进程消耗的 CPU 时间百分比的粗略计算。



可以在命令行中使用很多 ps 选项。如果要显示与第 3 个虚拟终端相关的所有进程,则可以使用 ps -ft /dev/pts/3。同样地, ps -fu oracle 只显示用户 oracle 拥有的进程。

-F 标志可以给出关于进程的更多细节。每列表示的信息在 ps 手册页中都有描述,并增加了 SZ、RSS 与 PSR 这 3 列。SZ 表示整个进程的页数目(在 x86 中通常是 4KB); RSS 表示进程占用的全部物理内存(不包括交换数据); PSR 表示进程运行所在的当前 CPU 的 ID。

```
$ ps -fp 3010
UID      PID PPID C   SZ   RSS PSR STIME TTY     TIME  CMD
Mysql    3010 2973 0 40228 22184 0 Oct24 ?    00:11:23 /usr/sbin/mysqld --
basedir=/usr --datadir=/var/lib/mysql --user=mysql --pid-file=/var/run/my
sqld/mysqld.pid -skip-external-loc king --port=3306
```

10.1.1 ps 显示的行宽

ps 会将输出的长度限制在终端的宽度以内。当写入对象不是终端(tty)时,则不会限制输出的宽度,而会显示整个命令行。这意味着脚本将得到输出的完整细节。例如, ps -fp 18611 | grep jrockit 会得到完整命令行。如果使用交互式终端进行某些测试, ps -fp 18611 可能不会在命令行部分显示进程 18611 的 jrockit 这样一个可见部分,即使有这一部分内容。

然而,运行 ps -fp 18611 | cat 能保证 ps 本身运行在管道中(不会输出到 tty),而且这也可能是使用 | cat 语法的唯一合理理由。如图 10-1 的截图所示,第一个 ps 命令连显示 java 可执行程序的完整目录路径的空间都没有,更不用提调用运行的是哪个程序。第二个 ps 命令显示了整个命令行,因为尽管输出最终被发送到终端设备,但 ps 发现其输出是连接到一个管道而不是 tty,所以显示完整的命令。



图 10-1

10.1.2 精确分析进程表

在 SysV 风格中, `ps -ft <terminal>` 与 `ps -fu <user>` 是从内核进程表中分别得到每个终端与每个用户记录的最好方法。然而, 通过 `ps` 命令的选项并不能对所有内容进行过滤。因此需要更有创造性的技术。随着时间的发展, 已经找到了一些不是 100%有效的方法, 所以还会有一些更新的命令被发明出来用于更加准确地分析进程表。



对于自动关闭进程脚本, 能精确识别目标进程极其重要, 尤其是当脚本是以 `root` 用户身份运行时。

通常, 要查找(例如)所有 Apache Web 服务器的进程, 我们可以运行:

```
$ ps -eaf | grep -w apache
```

但有时会得到以下(意料之外的)结果:

```
root      1742      1 0 19:46 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  1757 1742 0 19:46 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  1758 1742 0 19:46 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  1759 1742 0 19:46 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  1760 1742 0 19:46 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  1761 1742 0 19:46 ?        00:00:00 /usr/sbin/apache2 -k start
steve     2613 2365 0 20:11 pts/0    00:00:00 grep apache
```

结果包含了 `grep` 命令本身——`grep` 命令行包含了单词 `apache`。通过运行这个查找命令, 我们已经改变了进程表! 对于上面情况的传统解决方案是下面这个命令:

```
ps -eaf | grep -w apache | grep -v grep
```

这一命令是可行的, 除非 `grep` 刚好是要查找的 `ps` 输出的一部分(如果 Web 服务器用于进度报告并以名为 `progrep` 的用户身份运行, 则所有进程都会被忽略)。另一个传统且稍巧妙一些的方法是像下面的命令一样使用正则表达式:

```
ps -eaf | grep -w ap[a]che
```

这条命令会匹配 `apache` 而不会匹配命令本身! 这样做的缺点是它还会匹配所有名为 `apche` 的进程。如果要自动关闭匹配进程, 最好要确保没有匹配任何其他进程。

Linux(与现代 Unix)具备一套有用的命令集。这些命令能基于各种准则来识别进程。与对 `ps` 的输出执行 `grep` 不同, `pgrep` 默认只匹配实际的进程名。也就是说, 我们不会意外地匹配上例中以 `progrep` 用户身份运行的进程(`pgrep` 的 `-f` 标志确实可以匹配完整的命令行, 但很少用)。因为 `pgrep` 只返回一个 PID 列表, 因此在关闭脚本中, 我们可以运行下面的代码(`-x` 表示必须做精确匹配):

```
kill -9 `pgrep -x apache2`
```


而 `pgrep` 甚至更加灵活。我们可以指定只需要某些用户。所以如果某个程序有以不同用户 ID 运行的实例，我们可以只对一个用户进行识别，如使用 `pgrep -u devtest iidbms` 来列出 `devtest` 用户运行的 Ingres 数据库实例。还存在其他的方法，可以使用 `pgrep -u devtest` 查找属于某个用户的所有进程而不用管进程名称是什么。

因为 `pgrep` 经常用于关闭进程，所以它被符号链接到一个非常相似的命令 `pkill`。`pkill` 的语法与 `pgrep` 几乎完全一样，但不仅仅是列出 PID，它还会关闭进程。所以，之前的 `kill -9` 与 `pkill -x apache2` 相等。可以让 `pkill` 使用某个不同的信号，语法与 `kill` 相同：`pkill -l -x apache2` 将向进程发送信号 1。

表 10-1 列出了向进程发送的最常用信号。

表 10-1 常用信号

编 号	信 号	作 用
0	0	从 shell 退出
1	SIGHUP	清理；重新读取配置文件，然后继续运行
2	SIGINT	中断
3	SIGQUIT	退出
6	SIGABRT	中止
9	SIGKILL	立即关闭进程
14	SIGALRM	报警时钟
15	SIGTERM	清理并终止



当关闭机器时，OS 会正常调用注册过的关闭脚本，并向余下的进程发送 SIGTERM 信号，最后向仍然在运行的所有进程发送 SIGKILL 信号。

`pgrep` 还有很多其他标志，在手册页中都有文档说明。一个主要用于输出人类能理解的信息的有用标志是 `-l`，它能将进程名包含进来。下面的代码查找名称中包含 `ii` 的所有进程，并且不仅仅是获取 PID 列表，还包括进程名称。

```
$ pgrep -l ii
8402 iimerge
8421 iigcc
8376 iimerge
8439 iigcd
8387 iimerge
8216 iigcn
```

另一个常用的 `pgrep` 标志是指定定界符的 `-d` 开关。正常情况下，每个 PID 会显示在单独的一行。使用 `-d ''` 可以像 `pidof` 一样使用空格作为定界符，或者使用 `-d ','` 用逗号分隔。注意，PID 的顺序在这两个命令中是相反的。

```
$ pidof apache2
2510 2509 2508 2507 2502 1536 1535 1534 1533 1532 1495
$ pgrep -d',' apache2
1495,1532,1533,1534,1535,1536,2502,2507,2508,2509,2510
$
```

10.2 killall

一个有用的关闭所有进程的简单命令是 `killall`。`killall` 会关闭匹配给定条件的进程。与 `pgrep` 一类的命令非常相似，`killall` 也非常灵活。首先要注意的是，我们应当总是使用 `-e`(精确匹配)选项来运行 `killall`，除非确实知道所要做的以及明确了解系统中可能运行的进程。另一个常用选项是 `-u`，用来指定用户 ID 以对查找进行限制。如果 `apache` 进程以 `www` 用户身份运行，那么指定 `-u www` 可以确保不会对其他用户的进程造成影响。这与上例中的 `pgrep` 一样。`killall -u www` 本身会关闭 `www` 用户运行的每个进程——这与 `pgrep` 又是一样的。我们可以使用 `-s` 开关来指定不同的信号——所以 `killall -l -u www` 将向 `www` 的进程发送 `SIGHUP` 信号，要求进程重启。

下面是一个简单的应用程序的启动/关闭脚本：

```
$ cat /etc/init.d/myapp
#!/bin/bash
case "$1" in
    "start") su - myapp /path/to/program/startall
        # This may spawn a load of processes, whose names and PIDs may not be known.
        # However, unless suid is involved, they will be owned by the myapp user
        ;;
    "stop") killall -u myapp
        # This kills *everything* being run by the myapp user.
        ;;
    *) echo "Usage: `basename $0` start|stop"
        ;;
esac
$
```

很多服务会打开一组进程。系统管理员可能意识不到，这些进程名称可能会随着应用程序的版本或其他而发生变化。情况也可能是这样：应用程序用户以 `myapp` 登录，然后运行一些交互式 `shell`，并且不希望在关闭服务后关闭这些 `shell`。

如果已知应用程序所拥有的所有主要进程包含“`myapp_`”(`myapp_monitor`、`myapp_server` 和 `myapp_broker` 等)，那么通过指定 `killall -u myapp myapp_`，该脚本可以显得更加精细。

在 Unix(非 GNU/Linux)系统中要注意——`killall` 会关闭系统中的每个进程(不包括内核任务及其父进程)。GNU/Linux 为这一功能提供了命令 `killall5`。`killall5` 与 `pidof` 是同一个程序。`pgrep` 只查找二进制文件本身的名称，而 `pidof` 通过 ID 查找。如果 `Apache` 在进程表中以 `/usr/sbin/apache2` 的形式表示，则 `pidof apache2` 与 `pidof /usr/sbin/apache2` 都能成功匹配，

而 `pgrep /usr/sbin/apache2` 则没有任何结果。相反地, `gnome-terminal` 在进程表中没有路径, `pgrep` 无法找到 `/usr/bin/gnome-terminal`, 而 `pidof` 在两种情况下都能成功匹配。

```
$ ps -eaf | egrep "(apache|gnome-terminal)"
root      1476      1 0 18:36 ?        00:00:00 /usr/sbin/apache2 -k start
www-data 1580 1476 0 18:36 ?        00:00:00 /usr/sbin/apache2 -k start
www-data 1581 1476 0 18:36 ?        00:00:00 /usr/sbin/apache2 -k start
www-data 1582 1476 0 18:36 ?        00:00:00 /usr/sbin/apache2 -k start
www-data 1583 1476 0 18:36 ?        00:00:00 /usr/sbin/apache2 -k start
www-data 1584 1476 0 18:36 ?        00:00:00 /usr/sbin/apache2 -k start
steve     2461      1 0 18:37 ?        00:00:02 gnome-terminal
$ pgrep /usr/sbin/apache2
$ pgrep apache2
1476
1580
1581
1582
1583
1584
$ pidof /usr/sbin/apache2
1584 1583 1582 1581 1580 1476
$ pidof apache2
1584 1583 1582 1581 1580 1476
$ pgrep /usr/bin/gnome-terminal
$ pgrep gnome-terminal
2461
$ pidof gnome-terminal
2461
$ pidof /usr/bin/gnome-terminal
2461
$
```

10.3 /proc 虚拟文件系统

内核进程表与其中进程的状态都可以通过其 PID 从 `/proc` 虚拟文件系统中得到。如果需要关于 PID 28741 的信息, 则 `/proc/28741/` 中就包含了相关内容。还有一个特殊的符号链接 `/proc/self`: 对于任何引用 `/proc/self` 的进程, 该链接都会表现为指向运行进程的符号链接。这一点并不总是很明显:

```
$ echo $$
2168
$ ls -ld /proc/self /proc/$$
dr-xr-xr-x 7 steve steve 0 Nov 12 16:06 /proc/2168
lrwxrwxrwx 1 root root   64 Nov 12 15:56 /proc/self -> 2171
```

上面所完成的工作不是很直观。shell 的 PID 为 2168。shell 将 `$$` 的值传递给 `ls`。在 `ls` 程序中, `/proc/self` 是 `/proc/2171`, 因为 `ls` 运行的 PID 为 2171。所以这两个值不同。

下面的脚本使用 `/proc` 虚拟文件系统得到了一个给定进程与其最近运行所在 CPU 的状态信息。这很有用，例如可以与 `top` 列出的 I/O 等待状态信息进行关联。当然，我们可以用它来显示关于进程的几乎任何信息。在 Linux 内核源代码中，`/fs/proc/array.c` 包含 `do_task_stat()` 函数，它用来写 `/proc/<pid>/stat`。



可从
wrox.com
下载源代码

```
$ cat stat.sh
#!/bin/sh
# Example on RHEL5 (2.6.18 kernel):
#23267 (bash) S 23265 23267 23267 34818 23541 4202496 3005 27576 1 6 4 3 45 16 15 0
1 0 1269706754 72114176 448 18446744073709551615 4194304 4922060 140734626075216
18446744073709551615 272198374197 0 65536 3686404 1266761467 18446744071562230894 0
0 17 2 0 0 23
PID=${1}
if [ ! -z "$PID" ]; then
    read pid tcomm state ppid pgid sid tty_nr tty_pgrp flags min_flt cmin_flt
    maj_flt cmaj_flt utime stime cutime cstime priority nice num_threads
    it_real_value start_time vsize mm rsslim start_code end_code start_stack eis eip
    pending blocked sigign sigcatch wchan oul1 oul2 exit_signal cpu rt_priority
    policy ticks < /proc/$PID/stat
    echo "Pid $PID $tcomm is in state $state on CPU $cpu"
fi
```

stat.sh

脚本的显示结果如下：

```
Pid 2076 (bash) is in state S on CPU 1
```

由于 Linux 内核的开发模型，它提供的 API 并不保证 `/proc/*/stat` 不会随着时间变化。然而，要找到相关文档或是对 `/proc` 中文件进行写操作的内核部分，并进行一些修改，也并非难度很大。实际上，像 `/proc/*/stat` 这样的 API 会添加新的值以尽量保持兼容。例如，`it_real_value` 从 2.6.17 内核中移除掉后，它被替换为 0 (参见 <http://lkml.org/lkml/2006/2/14/312>)，这样 `start_time` 与它之后的字段会保持在与以前位置相同的地方。

可以调整输出，让结果显示虚拟文件 `/proc/$$/stat` 中的任何变量。例如：

```
echo "Pid $PID $tcomm is in state $state on CPU $cpu. Its parent is Pid $ppid."
echo "It is occupying $vsize bytes."
```

显示结果类似于下面的输出：

```
Pid 3053 (bash) is in state S on CPU 1. Its parent is Pid 2074.
It is occupying 19763200 bytes.
```

10.4 prtstat

实用程序 `prtstat` 是一个非常有用的工具。它像前面的脚本一样提供相同的信息，但却是一个外部的二进制文件。该程序是 `psmisc` 项目的一部分。`psmisc` 项目还提供了 `fuser`、

killall 与 pstree 命令。通过管道将 prtstat 连接到 grep 与 awk 可以得到 CPU 的编号，这比正确分析相对较长且复杂的变量集合要更容易。尽管结果看起来确实有些不美观，并且涉及产生 3 个不同的二进制文件，而上一节的 shell 脚本没有产生任何进程。

```
$ prtstat $$
Process: bash          State: S (sleeping)
CPU#: 1               TTY: 136:1   Threads: 1
Process, Group and Session IDs
  Process ID: 2168      Parent ID: 2063
  Group ID: 2168        Session ID: 2168
  T Group ID: 2999

Page Faults
  This Process      (minor major):      4539      0
  Child Processes   (minor major):      114675     55

CPU Times
  This Process      (user system guest blkio):  0.07   0.14   0.00   0.93
  Child processes   (user system guest):        4.56   0.67   0.00

Memory
  Vsize:            19 MB
  RSS:              2412 kB                RSS Limit: 18446744073709 MB
  Code Start:       0x400000              Code Stop: 0x4d8c1c
  Stack Start:      0x7fffb27ae720
  Stack Pointer (ESP): 0x7fffb27ae300    Inst Pointer (EIP): 0x7fe37fa9a36e

Scheduling
  Policy: normal
  Nice:    0          RT Priority: 0 (non RT)

$
$ prtstat -r $$ | grep processor: | awk '{ print $2 }'
0
```

10.5 I/O 重定向

每个进程在创建时都会打开 3 个标准文件。它们分别称为标准输入、标准输出与标准错误，文件描述符分别为 0、1 与 2。通常人们称它们为 stdin、stdout 与 stderr。ls-l /proc/self/fd 显示 ls 命令自身打开的文件。在标准的 0、1 与 2 之后，ls 还用文件描述符 3 打开了目录/proc/5820/fd 来列出它(5820 是 ls 自身的 PID)。

```
$ ls -l /proc/self/fd
total 0
lrwx----- 1 steve steve 64 Jan 27 21:34 0 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 27 21:34 1 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 27 21:34 2 -> /dev/pts/1
lr-x----- 1 steve steve 64 Jan 27 21:34 3 -> /proc/5820/fd
$
```

因为一切皆文件，所以连输入与输出也是文件。本例中的控制终端(/dev/pts/1)是输入

源，并且是输出与错误必须重定向的目标。ls 实际上不接收交互式输入，但其输出与任何必须报告的错误都流向控制终端，以便当用户运行 ls 命令时能在终端中看到输出。

因为它们都是文件，所以可以被重定向至其他文件。>操作符用来将输出从一个文件重定向至另一个文件。再次运行 ls 命令，但将输出重定向至/tmp/ls-output.txt，这时终端中没有任何显示。这是因为所有内容都输出到/tmp 中的文件。

```
$ ls -l /proc/self/fd > /tmp/ls-output.txt
$
```

通过输出 ls-output.txt 文件的内容，我们发现 ls 命令实际的显示中有个有趣的变化。标准输入(0)与标准错误(2)都还链接到/dev/pts/1，但标准输出(1)现在已经指向了 ls 在运行时创建的文件。

```
$ cat /tmp/ls-output.txt
total 0
lrwx----- 1 steve steve 64 Jan 27 21:42 0 -> /dev/pts/1
l-wx----- 1 steve steve 64 Jan 27 21:42 1 -> /tmp/ls-output.txt
lrwx----- 1 steve steve 64 Jan 27 21:42 2 -> /dev/pts/1
lr-x----- 1 steve steve 64 Jan 27 21:42 3 -> /proc/5839/fd
$
```

可以更进一步，使用语法 2>将标准错误(2)重定向至一个不同的文件。将 ls 的标准错误文件重定向至/tmp/ls-err.txt 会导致 ls 标准输出的进一步变化。



2 与>之间不能有空格，否则 shell 会将 2 解释成 ls 的参数，且将>解释成标准输出重定向。

```
$ ls -l /proc/self/fd > /tmp/ls-output.txt 2> /tmp/ls-err.txt
$ cat /tmp/ls-output.txt
total 0
lrwx----- 1 steve steve 64 Jan 27 21:50 0 -> /dev/pts/1
l-wx----- 1 steve steve 64 Jan 27 21:50 1 -> /tmp/ls-output.txt
l-wx----- 1 steve steve 64 Jan 27 21:50 2 -> /tmp/ls-err.txt
lr-x----- 1 steve steve 64 Jan 27 21:50 3 -> /proc/5858/fd
$ cat /tmp/ls-err.txt
$
```

/proc/self 能很好地演示重定向的工作原理。我们甚至还可以对标准输入(0)进行重定向，并特意使 ls 产生错误与输出。标准输出流向 ls-output.txt，而关于不存在文件的错误流向 ls-err.txt。

```
$ ls -l /proc/self/fd /nosuchfile > /tmp/ls-output.txt 2> /tmp/ls-err.txt
$ cat /tmp/ls-output.txt
/proc/self/fd:
total 0
lrwx----- 1 steve steve 64 Jan 27 21:54 0 -> /dev/pts/1
```

```

l-wx----- 1 steve steve 64 Jan 27 21:54 1 -> /tmp/ls-output.txt
l-wx----- 1 steve steve 64 Jan 27 21:54 2 -> /tmp/ls-err.txt
lr-x----- 1 steve steve 64 Jan 27 21:54 3 -> /proc/5863/fd
$ cat /tmp/ls-err.txt
ls: cannot access /nosuchfile: No such file or directory
$

```

最后的测试使用了某些特定输入。`ls` 非常适合于这种测试，它不从标准输入读取，但依然可以从别处读取输入。如果不管是否有意义，从语法上而言，将`/etc/hosts` 中的内容发送给 `ls` 也是合法的。最好首先用`</etc/hosts` 进行导向，然后用`>` 重定向标准输出，最后使用 `2>` 重定向标准错误。对于其他组合顺序，可能出现比较复杂的情况，会产生无法预料的结果。

```

$ ls -l /proc/self/fd /nosuchfile < /etc/hosts \
> > /tmp/ls-output.txt 2> /tmp/ls-err.txt
$ cat /tmp/ls-output.txt
/proc/self/fd:
total 0
lr-x----- 1 steve steve 64 Jan 27 22:52 0 -> /etc/hosts
l-wx----- 1 steve steve 64 Jan 27 22:52 1 -> /tmp/ls-output.txt
l-wx----- 1 steve steve 64 Jan 27 22:52 2 -> /tmp/ls-err.txt
lr-x----- 1 steve steve 64 Jan 27 22:52 3 -> /proc/2623/fd
$ cat /tmp/ls-err.txt
ls: cannot access /nosuchfile: No such file or directory
$

```

到目前为止，在 `ls` 命令打开的文件中都没有提及终端设备。所有的输入与输出都重定向至其他文件。当除了`/proc/self/fd` 以外的文件重定向时，这些重定向都位于`/proc` 虚拟文件系统外部。

10.5.1 向已有文件追加输出

`>` 语法创建长度为 0 的目标文件。如果不覆盖而是追加，则要使用`>>`。这样可以保留已有内容。但如果文件不存在，则会创建新文件。

```

$ cat names.txt
Homer
Marge
Bart
$ echo Lisa >> names.txt
$ cat names.txt
Homer
Marge
Bart
Lisa
$ echo Lisa > names.txt
$ cat names.txt
Lisa
$

```



>语法的另一个用法是有意地将已有文件清空。清理整个文件系统时的常见错误是删除一个正在被某个应用程序使用的日志文件。直到应用程序关闭文件，文件系统才会释放空间。使用>来截短文件可以使得文件大小变为 0，而且应用程序不必将其关闭。

10.5.2 重定向的权限

回过头来看看之前链接的权限，我们会发现它们不是标准的符号链接。符号链接一般的权限是 777——让符号链接有权限或具有自己的属主没有实际意义。因为内核会在重定向时强加某种语义上的权限，所以这些链接的权限会受到限制：与常规终端一样，标准输入不可写，标准输出或标准错误不可读。10.6 节将对这些内容进行深入介绍。

以只读形式打开的文件的权限为 r-x。打开用于写操作的文件的权限为 -rx，而打开用于读写操作的文件的权限为 rwx。与标准一致，不会有任何组或其他权限存在。

10.6 exec

内置命令 `exec` 调用内部的系统调用 `exec(3)`。该命令有两个主要目的。首先，它是系统调用的最常用方式——将当前运行进程替换为另一个进程。也就是说，shell 本身将被另一个程序替换，用来替换的可能是另一个 shell 或任何其他程序。当用 `exec` 调用的程序结束后，控制不会返回到调用 shell。其次，调用 `exec` 可以用来以副作用的形式实现重定向。

10.6.1 使用 `exec` 替换已有程序

下面是一个典型的登录会话，其中用户从 `node1` 登录到 `node2`。shell 提示符反映出了主机名。%提示符说明是 `csh` 会话，而\$提示符说明是 `bash` 会话。

```
steve@node1:~$ ssh node2 ←—— 用户从 node1 登录到 node2，提示符为 csh 提示符。
steve@node2's password:
You have new mail.
Last login: Mon Jan 17 15:19:53 2011
steve@node2%
steve@node2% bash ←—— 由于更喜欢 bash，用户从 csh 提示符调用了 bash。
steve@node2:~$ # do stuff in bash ←—— 随后，用户在 bash shell 中执行各种操作。
steve@node2:~$ echo $SHELL
/bin/bash
steve@node2:~$ exit ←—— 完成任务后，用户退出 shell。用户没有关闭连接，
exit                                     而只是退回到 csh 会话中。对于 csh 而言，bash 只
steve@node2%                               是另一个运行结束的程序而已。
steve@node2 % exit ←—— 用户退出 csh 会话，最后登出 node2。
logout
Connection to node2 closed.
steve@node1:~$
```

当使用内置命令 `exec` 时，用户不会返回到运行 `bash` 的 `ssh` 会话。原始的 `ssh` 进程被 `bash` 完全取代。

```

steve@node1:~$ ssh node2
steve@node2's password:
You have new mail.
Last login: Mon Jan 31 11:23:43 2011
steve@node2% exec bash ← csh 进程被一个 bash 进程替换。
steve@node2:~$ # do stuff in bash
steve@node2:~$ echo $SHELL
/bin/bash
steve@node2:~$ exit ← 当用户退出 bash shell 后，就没有可以回退的 csh 会话了。
exit                               因此到 node2 的连接被关闭了。
Connection to node2 closed.
steve@node1:~$

```

下面的会话简单地通过使用 `execs` 调用 `uname` 来进行一下总结。这与命令 `ssh node2 uname -a` 是等价的(至于 `node2`，其实现几乎是一模一样的)。运行结果显示，`node1` 的内核版本是 2.6.32-5，而 `node2` 的内核版本是 2.6.26-2。

```

steve@node1:~$ uname -a
Linux node1 2.6.32-5-amd64 #1 SMP Fri Dec 10 15:35:08 UTC 2010 x86_64 GNU/Linux
steve@node1:~$ ssh node2
steve@node2's password:
You have new mail.
Last login: Mon Jan 31 11:26:14 2011
steve@node2% exec uname -a
Linux node2 2.6.26-2-amd64 #1 SMP Sun Jun 20 20:16:30 UTC 2010 x86_64 GNU/Linux
Connection to node2 closed.
steve@node1:~$

```

还可以像下面这样以一行 `ssh` 命令来显示。

```

steve@node1:~$ uname -a
Linux node1 2.6.32-5-amd64 #1 SMP Fri Dec 10 15:35:08 UTC 2010 x86_64 GNU/Linux
steve@node1:~$ ssh node2 uname -a
Linux node2 2.6.26-2-amd64 #1 SMP Sun Jun 20 20:16:30 UTC 2010 x86_64 GNU/Linux
steve@node1:~$

```

10.6.2 使用 `exec` 修改重定向

内置命令 `exec` 的第二个用法是修改当前运行 `shell` 的重定向设置。`exec` 的这一用法更有趣。虽然这看起来非常晦涩与令人困惑，但如果正确理解了管道，这实际上是非常明确的。因为大多数使用这一技术的脚本本身一般都相当复杂，所以正确调试有时很困难。

理解 `exec` 重定向工作原理的最好方法就是使用一些非常简单的例子。这些脚本除了对一些测试文件进行 `exec` 外不会有任何更复杂的操作。但有时还是令人困惑，并且 `exec` 不

是在每个情况下都会按照预期的方式运行。这些脚本对 `/proc/$$/fd` 目录进行查看，显示运行的 shell 当前打开的(符号链接形式的)文件。

```
$ ls -l /proc/$$/fd
total 0
lrwx----- 1 steve steve 64 Jan 31 11:56 0 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 1 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 2 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:58 255 -> /dev/pts/1
$
```

打开的文件有 0(stdin)、1(stdout)与 2(stderr)。此处的 255 用到了一个小技巧。bash 用这一技巧保留对重定向的文件的一份副本。这是 bash 特有的功能。其他 shell 对于这些测试的运行方式与 bash 一样，但不会有文件描述符 255。

stdin、stdout 与 stderr 这 3 个名称在/dev 文件系统中有所体现。它们分别是对 `/proc/self/fd/0`、`/proc/self/fd/1` 与 `/proc/self/fd/2` 的符号链接。这只是 `/proc/self` 总是指向当前运行进程所带来的很多便利之处其中之一。带来这一便利性的原因在于，`/proc` 文件系统实际上是一个与正在运行的内核进行的交互式会话，而不仅仅是一个显示内核数据的文件系统。

```
$ ls -l /dev/std*
lrwxrwxrwx 1 root root 15 Jan 31 08:17 /dev/stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root 15 Jan 31 08:17 /dev/stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root 15 Jan 31 08:17 /dev/stdout -> /proc/self/fd/1
$
```

1. 打开用于写操作的文件

当运行 `exec 3> /tmp/testing` 时，shell 会创建新的指向 `/tmp/testing` 的文件描述符。它还会创建文件 `/tmp/testing`，并打开用于写操作。



3 与 > 之间没有空格，否则命令会被解释为向 exec 发出一个称为 3 的命令，并将标准输出重定向至 `/tmp/testing`。3>(还有任何有 > 或 < 紧随其后的数字)是 shell 的这种语法的一部分。

```
$ exec 3> /tmp/testing
$ ls -l /proc/$$/fd
total 0
lrwx----- 1 steve steve 64 Jan 31 11:56 0 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 1 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 2 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:58 255 -> /dev/pts/1
l-wx----- 1 steve steve 64 Jan 31 11:56 3 -> /tmp/testing
$
```


stdin、stdout 与 stderr 的运行结果与以前没有区别，但命令 `echo hello >&3` 被重定向至 3 号文件/tmp/testing。文件/tmp/testing 也能够正常读取——因为它只是一个文件。

```
$ echo hello
hello
$ echo hello >&3
$ cat /tmp/testing
hello
$ echo testing >&3
$ cat /tmp/testing
hello
testing
$
```

连续对&3 执行写操作会对文本进行追加，而不是覆盖。这与用/tmp/test-two 代替&3 的情况不同，因为对于/tmp/test-two 的单个>每次会重新创建文件。向文件描述符进行写操作更像是对网络设备或者打印机进行写操作。一旦数据被发送至目标，就不能像文件系统一样对文件进行截短。要对常规文件进行追加，则必须使用>>。

```
$ echo hello > /tmp/test-two
$ cat /tmp/test-two
hello
$ echo testing > /tmp/test-two
$ cat /tmp/test-two
testing
$
$ echo hello > /tmp/test-two
$ echo testing >> /tmp/test-two
$ cat /tmp/test-two
hello
testing
$
```

2. 打开用于读操作的文件

现在可以对文件描述符进行写操作，但输入与输出也同样重要。使用<可以使数据流反向。语法 `exec 4< /tmp/testing` 会创建一个文件描述符 4，并指向/tmp/testing，但打开的文件只能读取不能写入。

```
$ exec 4< /tmp/testing
$ ls -l /proc/$$/fd
total 0
lrwx----- 1 steve steve 64 Jan 31 11:56 0 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 1 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 2 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:58 255 -> /dev/pts/1
l-wx----- 1 steve steve 64 Jan 31 11:56 3 -> /tmp/testing
```

```
lr-x----- 1 steve steve 64 Jan 31 12:05 4 -> /tmp/testing
$ cat /tmp/testing
hello
testing
$ cat <&4
hello
testing
```

3. 跟踪文件位置

对常规文件进行读取与对文件描述符进行读取之间存在差异。我们可以对常规文件进行任意多次的读取，并且其内容不会发生变化。对文件描述符读取会消耗输入，就像 `read` 命令在 `while` 循环中运行时不断消耗输入。

```
$ cat /tmp/testing
hello
testing
$ cat <&4
$
```

进一步对文件描述符进行读写操作可以演示这一效果。将文件描述符看成流而不是实际的文件内容本身，这应当可以使概念更加清晰。



可以将 Unix 内部隐舍地比作水流。数据从一个位置流向另一个位置，甚至通过管道传输。这一比喻还可以用于 `<`、`<<`、`>` 与 `>>` 符号。数据流按照指定的方向流过这些漏斗。

```
$ echo more testing >&3
$ cat /tmp/testing
hello
testing
more testing
$ cat <&4
more testing
$
```

向文件描述符 3 执行写操作会向 `/tmp/testing` 追加内容，但因为文件描述符 4 已经在第 3 行，所以 `cat <&4` 命令只能返回新的数据，而 `cat /tmp/testing` 每次调用时都会重新打开文件。运行的 `shell` 知道其在文件描述符 4 中的位置，并且总保持这一位置。无论是对文件 `/tmp/testing` 进行追加还是向文件描述符 3 写入，`shell` 都会这样做。

```
$ echo append to the file itself >> /tmp/testing
$ cat /tmp/testing
hello
testing
more testing
```

```

append to the file itself
$ cat <&4
append to the file itself
$

```

如果将文件系统中的文件清空并写入新的数据，即使文件被>重定向截短，文件描述符 4 依然会保持其位置。

```

$ echo new data to the file > /tmp/testing
$ cat /tmp/testing
new data to the file
$ cat <&4
$

```

随着写入文件的内容越来越多，文件的长度最终到达文件描述符 4 的位置。在这种情况下，使用 exec 可能会产生一些令人迷惑的、无法预期(可能是破坏数据)的结果。在下面的例子中，even more data 这几个单词最后使文件比它在截短之前更长。结果显示了额外的字符 ore data，且原来的描述符 4 已经赶上当前内容。

```

$ echo lots more data >> /tmp/testing
$ echo more and more data >> /tmp/testing
$ cat <&4
$ echo even more data >> /tmp/testing
$ cat <&4
ore data
$

```

如果文件被移除，会发生类似的破坏性结果。内核会对这进行跟踪，且/proc/\$\$/fd/列表中会显示单词“(deleted)”。

```

$ ls -l /proc/$$/fd
total 0
lrwx----- 1 steve steve 64 Jan 31 11:56 0 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 1 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 2 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:58 255 -> /dev/pts/1
l-wx----- 1 steve steve 64 Jan 31 11:56 3 -> /tmp/testing
lr-x----- 1 steve steve 64 Jan 31 12:05 4 -> /tmp/testing
$ rm /tmp/testing
$ ls -l /proc/$$/fd
total 0
lrwx----- 1 steve steve 64 Jan 31 11:56 0 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 1 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 2 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:58 255 -> /dev/pts/1
l-wx----- 1 steve steve 64 Jan 31 11:56 3 -> /tmp/testing (deleted)
lr-x----- 1 steve steve 64 Jan 31 12:06 4 -> /tmp/testing (deleted)
$

```

相比其他而言，这是文件系统的另一个重要用法。文件已经被删除(可能被系统管理员

删除, 原因是/tmp 文件系统已经没有空间, 而且/tmp/testing 是一个较大的日志文件), 但 bash 进程依然以打开文件的形式对其进行保留。无论文件的属主进程是 shell 脚本、Apache 服务器、Oracle 数据库还是任何其他程序, 都是如此。尽管文件已经从文件系统中删除, 但它依然存在于内核的文件系统驱动程序中, 直到最后一个进程关闭它。文件系统空间也还没有被释放。尽管显式的 ls 调用对于其他进程显示的是文件不存在, 但进程继续对文件进行读取, 就好像文件没有被删除一样。

```
$ ls -l /proc/$$/fd
total 0
lrwx----- 1 steve steve 64 Jan 31 11:56 0 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 1 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:56 2 -> /dev/pts/1
lrwx----- 1 steve steve 64 Jan 31 11:58 255 -> /dev/pts/1
l-wx----- 1 steve steve 64 Jan 31 11:56 3 -> /tmp/testing (deleted)
lr-x----- 1 steve steve 64 Jan 31 12:05 4 -> /tmp/testing (deleted)
$ echo line one >&3
$ echo line two >&3
$ echo line three >&3
$ cat <&4
line one
line two
line three
$ cat /tmp/testing
cat: /tmp/testing: No such file or directory
$ echo line four >&3
$ cat <&4
line four
$
```

所以显而易见的是, 系统管理员无法在不关闭进程(或者至少让进程关闭文件)的前提下释放空间。实际上, 管理员可以通过向文件进行写操作来释放文件系统空间。这在本节之前的文件描述符 4 的例子中可以看到。如果随后的循环持续向磁盘进行写操作, 那么它最终将占满整个文件系统。



在下面这个例子中, while 循环必须是使 date.log 打开的单一进程。否则, 如果每次循环都调用 date 打开文件并向其追加, 其行为可能会有不同。这样的典型应用程序进程将以这种方式使文件打开, 而不是每次要追加的时候打开。如果应用程序确实关闭了文件, 则很容易将其移动或删除。

```
$ while :
> do
>   date
>   sleep 1
> done >> date.log
```

随着时间的推移，硬盘最终被占满：

```
date: write error: No space left on device
date: write error: No space left on device
date: write error: No space left on device
date: write error: No space left on device
date: write error: No space left on device
date: write error: No space left on device
```

删除文件没有意义，因为如上面的代码所示，进程将使文件的副本保持打开。释放空间的方法是对文件进行写操作。

```
[root@server]# tail date.log
Mon Feb 7 12:46:23 GMT 2011
Mon Feb 7 12:46:24 GMT 2011
Mon Feb 7 12:46:25 GMT 2011
Mon Feb 7 12:46:26 GMT 2011
Mon Feb 7 12:46:27 GMT 2011
Mon Feb 7 12:46:28 GMT 2011
Mon Feb 7 12:46:29 GMT 2011
Mon Feb 7 12:46:30 GMT 2011
Mon Feb 7 12:46:31 GMT 2011
Mon Feb 7 12:46:32 G[root@server]#
[root@server]# df -k .
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/sdb1        616636      616636         0 100% /var/datalog
```

到此，文件系统被占满，甚至 12:46:32 那一行也没有完全显示出来。使用操作符>重置 date.log，释放 8 个磁盘区块。

```
[root@server]# > date.log
[root@server]# df -k .
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/sdb1        616636      616628         8 100% /var/datalog
[root@server]# date
Mon Feb 7 12:46:58 GMT 2011
[root@server]# cat date.log
Mon Feb 7 12:46:53 GMT 2011
Mon Feb 7 12:46:54 GMT 2011
Mon Feb 7 12:46:55 GMT 2011
Mon Feb 7 12:46:57 GMT 2011
Mon Feb 7 12:46:58 GMT 2011
Mon Feb 7 12:46:59 GMT 2011
Mon Feb 7 12:47:00 GMT 2011
Mon Feb 7 12:47:01 GMT 2011
Mon Feb 7 12:47:02 GMT 2011
Mon Feb 7 12:47:03 GMT 2011
Mon Feb 7 12:47:04 GMT 2011
[root@server]# wc -l date.log
21 date.log
[root@server]#
```

文件系统的空间得到释放，循环继续向文件写入，且不用重启。如 `date` 与 `wc` 命令所显示的，日志文件继续被实时地写入。

10.7 管道

管道是 Unix 与 Linux shell 的重要特性。管道将两个进程连接在一起，通常是将一个进程的标准输出连接到另一个进程的标准输入。管道并没有将第一个命令的输出写到一个文件，然后将该文件作为输入运行第二个命令。使用管道完全可以省去中间文件。对于 shell 脚本编程而言，管道的概念如此重要，以至于之前多次提到。然而理清管道的实际工作原理还是有意义的。下面的例子将 `find` 的输出用管道传递给 `grep`：

```
$ find / -print | grep hosts
/lib/security/pam_rhosts.so
/var/lib/ghostscript
/var/lib/ghostscript/CMap
/var/lib/ghostscript/fonts
/var/lib/ghostscript/fonts/cidfmap
/var/lib/ghostscript/fonts/Fontmap
find: `/var/lib/php5': Permission denied
find: `/var/lib/polkit-1': Permission denied
/var/lib/dpkg/info/denyhosts.postinst
```

上面的命令首先启动 `grep`，然后是 `find`。一旦有了这两个进程，`find` 进程的输出会被连接到 `grep` 进程的输入。`find` 进程随后会运行，并将其输出发送给 `grep`。如上面的输出所示，`find` 进程的标准错误设备依然是调用终端，所以显示正常。显示内容没有传递给 `grep`。这很显然，因为错误行中没有包含文本 `hosts`。因此，显示内容是 `find` 的标准错误，而不是 `grep` 的输出。

10.8 后台处理

有时让命令在后台运行并立即将控制返回 shell 会比较有用。如果脚本不依赖于执行命令的结果或状态，则没有必要等待其结束。命令行最后的 `&` 符号可以实现进程的后台运行。后台进程的 PID 被赋值给 `!`，然后主脚本或交互式 shell 继续照常执行。主脚本还可以对后台运行的子进程运行中的活动进行监视。下面的 `dd` 命令从 `/dev/urandom` 驱动程序中读取 512MB 的随机数据，然后写入到 `bigfile` 文件中。在 `bigfile` 增长的同时，交互式 shell 会显示它，并通过 `ps` 与 `strace` 使用 `!` 变量来监视这一过程。`strace` 的输出显示从 `/dev/urandom` 读取并写入到 `bigfile` 中的随机数据。

```
$ dd if=/dev/urandom of=bigfile bs=1024k count=512 &
[1] 3495
$ ls -lh bigfile
-rw-rw-r-- 1 steve steve 18M Jan 28 18:01 bigfile
```

```

$ ls -lh bigfile
-rw-rw-r-- 1 steve steve 196M Jan 28 18:02 bigfile
$ ps -fp $!
UID      PID PPID  C STIME TTY          TIME CMD
Steve 3495 3363 99 18:01 pts/1 00:00:44 dd if=/dev/urandom of=bigfile bs=10
24k count=512
$ strace -p $! 2>&1 | head -5
Process 3495 attached - interrupt to quit
write(1, "\251\0\322\335J\362\214\334\331\342\213\356\377\23%\371\353U\377H\262\225
'w\r`_\316\306\220\325g"... , 828440) = 828440
read(0, "znm9;\311)\344\21z\342\" \215n\272d8\24\321\215\363\340\327%\213\3623&\273;
\10\323"... , 1048576) = 1048576
write(1, "znm9;\311)\344\21z\342\" \215n\272d8\24\321\215\363\340\327%\213\3623&\273
;\10\323"... , 1048576) = 1048576
read(0, "\fq31\273\343c/\300\343\31\262V\263\222\351\310\310/\274t\223\330\217\223\
345H\221B\310\237\246"... , 1048576) = 1048576
$ ls -lh bigfile
-rw-rw-r-- 1 steve steve 288M Jan 28 18:02 bigfile
$ 509+3 records in
509+3 records out
53
[1]+  Done                  dd if=/dev/urandom of=bigfile bs=1024k count=512
$ rm bigfile
$

```

第 11 章将介绍交互式工具 **bg** 与 **fg**。它们是用于处理多进程的有用工具。**bg** 让停止的作业在后台继续执行，而 **fg** 则将后台作业转到前台。

10.8.1 wait 命令

可以等待一个、多个甚至全部的后台进程直到它们运行结束。下面的会话下载了一个非常小的 `md5sum.txt` 文件，然后下载 CentOS 5.5 的前两个 CD-ROM 的 ISO 映像。映像是大文件，所以在后台下载它们，以便让控制可以返回到 shell。

```

$ wget http://mirror.ox.ac.uk/sites/mirror.centos.org/5.5/isos/x86_64/md5sum.txt
--2011-02-12 22:27:46-- http://mirror.ox.ac.uk/sites/mirror.centos.org/5.5/isos/x8
6_64/md5sum.txt
Resolving mirror.ox.ac.uk... 163.1.2.224, 163.1.2.231
Connecting to mirror.ox.ac.uk|163.1.2.224|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 788 [text/plain]
Saving to: `md5sum.txt'

100%[=====>] 788                --.-K/s in 0s

2011-02-12 22:27:46 (50.8 MB/s) - `md5sum.txt' saved [788/788]

$ wget http://mirror.ox.ac.uk/sites/mirror.centos.org/5.5/isos/x86_64
/CentOS-5.5-x86_64-bin-1of8.iso > /dev/null 2>&1 &

```

```

[1] 4572
$ wget http://mirror.ox.ac.uk/sites/mirror.centos.org/5.5/isos/x86_64/
CentOS-5.5-x86_64-bin-2of8.iso > /dev/null 2>&1 &
[2] 4573
$ ps -f
UID      PID PPID C STIME TTY          TIME CMD
steve 4555 4554 0 22:26 pts/3 00:00:00 -bash
steve 4572 4555 2 22:27 pts/3 00:00:00 wget http://mirror.ox.ac.uk/site
steve 4573 4555 2 22:28 pts/3 00:00:00 wget http://mirror.ox.ac.uk/site
steve 4574 4555 0 22:28 pts/3 00:00:00 ps -f
$ jobs
[1]-  Running      wget http://mirror.ox.ac.uk/sites/mirror.centos.org/5
.5/isos/x86_64/CentOS-5.5-x86_64-bin-1of8.iso > /dev/null 2>&1 &
[2]+  Running      wget http://mirror.ox.ac.uk/sites/mirror.centos.org/5
.5/isos/x86_64/CentOS-5.5-x86_64-bin-2of8.iso > /dev/null 2>&1 &
$ wait

```

不可能在同一个终端将这两个活动的 `wget` 进程提到前台，但最后的 `wait` 命令有相似的效果。该命令会等待调用者所有的子进程返回，然后将控制返回给调用 `shell`。这对于交互式 `shell` 而言不是非常重要，但脚本可以使用这种方法在比较 MD5 校验和前获得全部的 ISO 映像，而且也只有在下载完成后才能比较 MD5 校验和。

```

[1]- Done          wget http://mirror.ox.ac.uk/sites/mirror.centos.org/5
.5/isos/x86_64/CentOS-5.5-x86_64-bin-1of8.iso > /dev/null 2>&1
[2]+ Done          wget http://mirror.ox.ac.uk/sites/mirror.centos.org/5
.5/isos/x86_64/CentOS-5.5-x86_64-bin-2of8.iso > /dev/null 2>&1
$ md5sum -c md5sum.txt
CentOS-5.5-x86_64-bin-1of8.iso: OK
CentOS-5.5-x86_64-bin-2of8.iso: OK
$

```



简单地说，`md5sum.txt` 被剪裁后只包含这两个文件。否则，会显示关于另外 6 个丢失的 ISO 映像 3~8 的错误消息。

10.8.2 使用 `nohup` 防止进程挂起

当运行一个像上一节中下载任务这样时间非常长的后台进程时，最好能保证即使用户登出其会话或因为网络链接断开而退出，作业也不会终止。`nohup` 命令将运行的进程封装起来，保护它不接收其他使其终止的信号。虽然它们不总是一起使用，但通常将 `nohup` 的命令置于后台运行。类似地，通常使用 `nohup` 运行后台进程。

默认情况下，`nohup` 的输出会写入到 `nohup.out`。然而，如果 `stderr` 与 `stdout` 都被重定向到别处，则不会创建 `nohup.out`。

下面的脚本试验了所有这些情况，作用是下载并自动验证 ISO 映像。下面的代码将日志输出到独立的文件，每个映像对应一个文件，最后执行适当的 MD5 校验和检查。



可从
wrox.com
下载源代码

```
$ cat getisos.sh
#!/bin/bash
MIRROR=http://mirror.ox.ac.uk/sites/mirror.centos.org/5.5/isos/x86_64
IMAGE=CentOS-5.5-x86_64-binwget

Wget ${MIRROR}/md5sum.txt > md5.out 2>&1
for image in ${IMAGE}{1,2,3,4,5,6,7,8}of8.iso
do
    nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
    grep ${image} md5sum.txt >> files-to-check.txt
done

echo "Waiting for files to download..."

jobs
wait
echo "Verifying MD5 sums..."
md5sum -c files-to-check.txt
if [ "$?" -eq "0" ]; then
    echo "All files downloaded successfully."
else
    echo "Some files failed."
    exit 1
fi
$
```

getisos.sh

```
$ ./getisos.sh
Waiting for files to download...
[1] Running nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
[2] Running nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
[3] Running nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
[4] Running nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
[5] Running nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
[6] Running nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
[7]- Running nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
[8]+ Running nohup wget ${MIRROR}/${image} > ${image}.out 2>&1 &
```

在并行下载 8 个 ISO 映像时会有很长一段延迟。最后，一旦所有的文件都被下载下来，它们都被 md5sum 实用程序检查。

```
Verifying MD5 sums...
CentOS-5.5-x86_64-bin-1of8.iso: OK
CentOS-5.5-x86_64-bin-2of8.iso: OK
CentOS-5.5-x86_64-bin-3of8.iso: OK
```

```
CentOS-5.5-x86_64-bin-4of8.iso: OK
CentOS-5.5-x86_64-bin-5of8.iso: OK
CentOS-5.5-x86_64-bin-6of8.iso: OK
CentOS-5.5-x86_64-bin-7of8.iso: OK
CentOS-5.5-x86_64-bin-8of8.iso: OK
All files downloaded successfully.
```

如果下载没有成功，则很容易看到特定下载的输出。在本例中，映像 6 下载失败。wget 的输出相当冗长，所以使用 **head** 检查下载成功开始，使用 **tail** 显示正常完成。



另外，有一些程序会检查输出何时不发送给终端，并且在这种情况下输出甚至更加冗长。wget 就是其中之一。wget 会在终端中显示一个进度条，但在重定向时会给出关于下载进度的更详细的消息。

```
Verifying MD5 sums...
CentOS-5.5-x86_64-bin-1of8.iso: OK
CentOS-5.5-x86_64-bin-2of8.iso: OK
CentOS-5.5-x86_64-bin-3of8.iso: OK
CentOS-5.5-x86_64-bin-4of8.iso: OK
CentOS-5.5-x86_64-bin-5of8.iso: OK
CentOS-5.5-x86_64-bin-6of8.iso: FAILED
CentOS-5.5-x86_64-bin-7of8.iso: OK
CentOS-5.5-x86_64-bin-8of8.iso: OK
md5sum: WARNING: 1 of 8 computed checksums did NOT match
Some files failed.
$ head CentOS-5.5-x86_64-bin-6of8.out
--2011-02-12 22:53:20-- http://mirror.ox.ac.uk/sites/mirror.centos.org/5.5/isos/x86_64/CentOS-5.5-x86_64-bin-6of8.iso
Resolving mirror.ox.ac.uk... 163.1.2.224, 163.1.2.231
Connecting to mirror.ox.ac.uk|163.1.2.224|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 657436672 (627M) [application/x-iso9660-image]
Saving to: `CentOS-5.5-x86_64-bin-6of8.iso'

      OK..... 0% 113K 94m42s
      50K..... 0% 223K 71m23s
      100K..... 0% 204K 65m6s
$ tail -4 CentOS-5.5-x86_64-bin-6of8.out
638200K..... 99% 141M 0s
638250K..... 99% 180M 0s
638300K..... 100% 173M=83s

2011-02-12 23:12:26 (115K/s) - `CentOS-5.5-x86_64-bin-6of8.iso' saved [653668352/653668352]
$
```

上面的结果显示文件似乎已成功下载，所以肯定是在 Internet 上传输的时候被损坏了。

对单个文件运行 md5sum 实用程序进行确认——校验和不匹配。

```
$ grep CentOS-5.5-x86_64-bin-6of8.iso md5sum.txt
f0b40f050e17c90e5dbba9ef772f6886 CentOS-5.5-x86_64-bin-6of8.iso
$ md5sum CentOS-5.5-x86_64-bin-6of8.iso
50ef685abe51db964760c6d20d26cf31 CentOS-5.5-x86_64-bin-6of8.iso
$
```

10.9 /proc 和/sys 的其他特性

如上文提到的，Linux 中的/proc 文件系统显示有关内核的大量信息。这些信息不是(刚开始可能是)启动时创建的一组静态的文件，而是直接与内核本身挂钩。相应地，可以向一些文件写入，可以从一些文件读取，另一些文件则可读可写。/proc 的一些特性在 proc(5) 手册页中有描述，可以运行 man 5 proc 查看。这些文件对于 shell 脚本非常有用，因为 shell 脚本可能与内核内部本身非常接近。但这在类 Unix 系统中并不常见。

10.9.1 /proc/version

/proc/version 是有关内核版本的只读列表，其中包含了如何编译的详细构建信息。它可以比 uname 更完整、更精确地检测实际内核版本。不管它看起来如何，实际上只是一行文本而已。

```
$ cat /proc/version
Linux version 2.6.32-5-amd64 (Debian 2.6.32-29) (ben@decadent.org.uk) (gcc version
4.3.5 (Debian 4.3.5-4) ) #1 SMP Fri Dec 10 15:35:08 UTC 2010
$ uname -a
Linux goldie 2.6.32-5-amd64 #1 SMP Fri Dec 10 15:35:08 UTC 2010 x86_64 GNU/Linux
$
```

10.9.2 SysRq

PC 键盘上有一个很少用的标记为 SysRq 的按键。其历史可以追溯到大型机系统，但 Linux 内核用它在常规方法(如很多例子中介绍的对/proc 中的文件进行回显操作)不可能的情况下与内核进行通信。如果启用这一功能，当用户按下组合键 Ctrl+Alt+SysRq 与另一个指定内核操作的按键时，则内核会完成一些提供好的最基本任务——同步到文件系统、报告内存使用情况或者重启系统。表 10-2 列出了这些可用功能。

表 10-2 常用 SysRq 命令

按 键	作 用
c	使系统崩溃
m	显示系统内存
h	显示帮助

(续表)

按 键	作 用
r	将控制台显示模式设置为 Raw
s	同步所有文件系统
i	向所有进程(除了 init)发送 KILL 信号
u	卸载所有文件系统
b	重启机器
e	向所有进程(除了 init)发送 TERM 信号

有一个快捷键记忆法，即 **R**aising **S**kinny **E**lephants **I**s **U**tterly **B**oring(译注：每个单词的首字母表示按键)。要安全重启一个挂起的系统，这些快捷键会将控制台设置为原始模式，同步文件系统，向所有进程发送 TERM 信号，卸载所有文件系统，然后重启机器。图 10-2 显示了控制台中在按下 Control-Alt-SysRq 与 h 的组合后显示 SysRq 帮助信息，随后使用 Control-Alt-SysRq 与 s 同步磁盘。

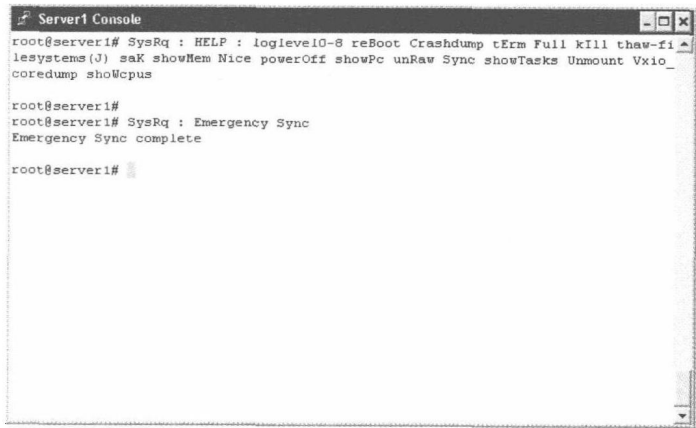


图 10-2

sysrq 特性在 /proc 下有两个文件：/proc/sysrq-trigger 和 /proc/sys/kernel/sysrq。后者启用或禁用 sysrq 特性，对于所有用户可读，对于 root 用户可写。1 表示特性被启用，0 表示特性被禁用。

/proc/sysrq-trigger 文件从表 10-2 中挑选一个按键，然后执行与用户在系统控制台中按下 Ctrl+Alt+SysRq 与那个按键组合一样的功能。这在控制台位于远程数据中心的情况下特别有用。

```
declan:~# echo h > /proc/sysrq-trigger
declan:~# tail -1 /var/log/messages
Feb 13 22:17:44 declan kernel: [2328070.124615] SysRq : HELP: loglevel0-8 reBoot
Crashdump tErm Full kIll saK aLlcpus showMem Nice powerOff showPc show-all-timers(Q
)
```

```

unRaw Sync showTasks Unmount show-blocked-tasks
declan:~# echo m > /proc/sysrq-trigger
declan:~# tail -24 /var/log/messages
Feb 13 22:17:52 declan kernel: [2328078.469396] SysRq : Show Memory
Feb 13 22:17:52 declan kernel: [2328078.469396] Mem-info:
Feb 13 22:17:52 declan kernel: [2328078.469396] Node 0 DMA per-cpu:
Feb 13 22:17:52 declan kernel: [2328078.469396] CPU 0: hi: 0, btch: 1 usd:
0
Feb 13 22:17:52 declan kernel: [2328078.469396] CPU 1: hi: 0, btch: 1 usd:
0
Feb 13 22:17:52 declan kernel: [2328078.469396] Node 0 DMA32 per-cpu:
Feb 13 22:17:52 declan kernel: [2328078.469396] CPU 0: hi: 186, btch: 31 usd:
81
Feb 13 22:17:52 declan kernel: [2328078.469396] CPU 1: hi: 186, btch: 31 usd:
140
Feb 13 22:17:52 declan kernel: [2328078.469396] Active:109030 inactive:364454
dirty:35 writeback:0 unstable:0
Feb 13 22:17:52 declan kernel: [2328078.469396] free:11420 slab:23854 mapped:
9902pagetables:1888 bounce:0
Feb 13 22:17:52 declan kernel: [2328078.469396] Node 0 DMA free:8376kB min:28kB
low:32kB high:40kB active:124kB inactive:3032kB present:10788kB pages_scanned:0
all_unreclaimable? no
Feb 13 22:17:52 declan kernel: [2328078.469396] lowmem_reserve[]: 0 2003 2003
2003
Feb 13 22:17:52 declan kernel: [2328078.469396] Node 0 DMA32 free:37304kB
min:5708kB low:7132kB high:8560kB active:435996kB inactive:1454784kB present:2051120kB
pages_scanned:0 all_unreclaimable? no
Feb 13 22:17:52 declan kernel: [2328078.469396] lowmem_reserve[]: 0 0 0 0
Feb 13 22:17:52 declan kernel: [2328078.469396] Node 0 DMA: 10*4kB 10*8kB 4*16kB
4*32kB 4*64kB 1*128kB 2*256kB 2*512kB 2*1024kB 0*2048kB 1*4096kB = 8376kB
Feb 13 22:17:52 declan kernel: [2328078.469396] Node 0 DMA32: 2090*4kB 2250*8kB
280*16kB 12*32kB 1*64kB 1*128kB 1*256kB 1*512kB 1*1024kB 0*2048kB 1*4096kB = 37304kB
Feb 13 22:17:52 declan kernel: [2328078.469396] 444427 total pagecache
pages
Feb 13 22:17:52 declan kernel: [2328078.469396] Swap cache: add 214, delete 13,
find 0/0
Feb 13 22:17:52 declan kernel: [2328078.469396] Free swap = 1951004kB
Feb 13 22:17:52 declan kernel: [2328078.469396] Total swap = 1951856kB
Feb 13 22:17:52 declan kernel: [2328078.469396] 523984 pages of RAM
Feb 13 22:17:52 declan kernel: [2328078.469396] 8378 reserved pages
Feb 13 22:17:52 declan kernel: [2328078.469396] 332107 pages shared
Feb 13 22:17:52 declan kernel: [2328078.469396] 201 pages swap cached
declan:~#

```

10.9.3 /proc/meminfo

/proc/meminfo 提供关于当前内存与虚拟内存系统状态的相当详细的信息。下面的代码突出了一些非常有趣的地方。Memtotal 与 MemFree 的含义可以顾名思义，它们分别是可供使用的物理内存与空闲的物理内存。虚拟内存子系统包括交换空间，也在这里显示出来，

另外还有文件系统缓冲区与高速缓存的当前状态。`/proc/meminfo` 是系统信息的有用来源，而且与大多数 `/proc` 中的文件类似，它被设计成很容易让人和脚本分析。10.9.6 节将给出一个稍高级一些的等价文件。

```
MemTotal:    2062424 kB
MemFree:      46984 kB
Buffers:      201248 kB
Cached:       1575452 kB
SwapTotal:    1951856 kB
SwapFree:     1951004 kB
```

10.9.4 `/proc/cpuinfo`

只读文件 `/proc/cpuinfo` 显示安装在系统中的处理器的信息。严格来说，它显示的是系统中的处理线程，所以超线程、多核与多处理器系统会列出很多 CPU。要查看系统中有多少物理 CPU 或一个 CPU 有多少个核则不是那么直观。`physical id` 字段对 CPU 核从 0 开始计数，所以如果最高的物理 ID 是 3，则机器中实际上有 4 个芯片。类似地，`core id` 字段对芯片中核的数目进行计数，所以如果一块 CPU 的核是从 0 到 5，则它是一个 6 核 CPU。超线程将可用的执行线程增倍，但超线程的核只在 `/proc/cpuinfo` 中出现一次。

`cpuinfo.sh` 脚本对 `/proc/cpuinfo` 进行分析，然后给出一个对系统处理器的人类可读的描述。



可从
wrox.com
下载源代码

```
$ cat cpuinfo.sh
#!/bin/bash
hyperthreads=1
grep -w "^flags" /proc/cpuinfo | grep -qw "ht" && hyperthreads=2
phys=`grep "physical id" /proc/cpuinfo | sort -u | wc -l`
cores=`grep "core id" /proc/cpuinfo | sort -u | wc -l`
threads=`expr $phys \* $cores \* $hyperthreads`
detail=`grep "model name" /proc/cpuinfo | sort -u | cut -d: -f2- \
| cut -c2- | tr ~s " " `
echo "`hostname -s` has $phys physical CPUs ($detail) each with $cores cores. "
echo "Each core has $hyperthreads threads: total $threads threads"
$ ./cpuinfo.sh
webserv has 2 physical CPUs (Intel(R) Xeon(R) CPU L5420 @ 2.50GHz) each with 4
cores.
Each core has 2 threads: total 16 threads
$
```

`cpuinfo.sh`

10.9.5 `/sys`

`/sys` 是与 `/proc` 密切相关的虚拟文件系统。它们之间甚至有一些重叠，因为 `/proc` 有一个包含类似内容的子目录 `/sys`。下面的 `cpu.sh` 脚本对 `/sys/devices/system/node` 树进行读写。本章后面的 `mem.sh` 脚本从一个相同的树中读取内存配置。



可从
wrox.com
下载源代码

```
$ cat cpu.sh
```

```
#!/bin/bash
```

```
if [ ! -f /sys/devices/system/node/node0/cpu0/online ]; then
    echo "node0/cpu0 is always Online."
fi
```

```
function showcpus()
```

```
{
    cpu=${1:-'*'}
    for node in `ls -d /sys/devices/system/node/node*/cpu${cpu} | \
        cut -d"/" -f6 | sort -u`
    do
        grep . /sys/devices/system/node/${node}/cpu*/online /dev/null \
            | cut -d"/" -f6- | sed s/"\//online"/""/g | \
            sed s/":1$"/" is Online"/g | sed s/":0$"/" is Offline"/g
    done
}
```

```
function online()
```

```
{
    if [ ! -f /sys/devices/system/node/node*/cpu${1}/online ]; then
        echo "CPU$1 does not have online/offline functionality"
    else
        grep -q 1 /sys/devices/system/node/node*/cpu${1}/online
        if [ "$?" -eq "0" ]; then
            echo "CPU$cpu is already Online"
        else
            echo -en "`showcpus $cpu` - "
            echo -en "Onlining CPU$cpu ... "
            echo 1 > /sys/devices/system/node/node*/cpu${1}/online 2> /dev/null
            if [ "$?" -eq "0" ]; then
                echo "OK"
            else
                echo "Failed to online CPU$1"
            fi
        fi
    fi
}
```

```
function offline()
```

```
{
    if [ ! -f /sys/devices/system/node/node*/cpu${1}/online ]; then
        echo "CPU$1 does not have online/offline functionality"
    else
        grep -q 0 /sys/devices/system/node/node*/cpu${1}/online
        if [ "$?" -eq "0" ]; then
            echo "CPU$cpu is already Offline"
        else
            echo -en "`showcpus $cpu` - "

```

```

echo -en "Offlining CPU$cpu ... "
echo 0 > /sys/devices/system/node/node*/cpu${1}/online 2> /dev/null
if [ "$?" -eq "0" ]; then
    echo "OK"
else
    echo "Failed to offline CPU$1"
fi
fi
fi
}

case $1 in
show) showcpus $2
    ;;
on)
    shift                # Lose the keyword
    for cpu in $*
    do
        online $cpu
    done
    ;;
off)
    shift                # Lose the keyword
    for cpu in $*
    do
        offline $cpu
    done
    ;;
*) echo "Usage: "
    echo " `basename $0` show [ cpu# ] - shows all CPUs shared by that node"
    echo " `basename $0` on cpu# ( cpu# cpu# cpu# ... )"
    echo " `basename $0` off cpu# ( cpu# cpu# cpu# ... )"
    ;;
esac

```

cpu.sh

该脚本的主体是一个 `case` 语句，它读取传递给脚本的第一个单词(`on`、`off` 与 `show` 是合法关键词)，然后按照需求调用相关函数。对于 `show`，关于 CPU id 的可选参数将输出限制在单个 CPU。如果不提供这一参数，则 `showcpus` 函数开头的 `cpu=${1:-'*'}` 将这一参数值设置为星号。星号在之后作为通配符匹配系统中的所有 CPU。



NUMA 是 Non-Uniform Memory Architecture 的缩写，它可以表示像 Beowulf 群集这样的大型机器群集。NUMA 也可以指多个多核 CPU，其中一部分系统 RAM 被绑定到一个 CPU，而其他内存地址空间则绑定到另一个 CPU。

对于 `on` 和 `off` 关键词，`shift` 命令可以将关键词本身从参数列表中移除，而只剩下 CPU

编号列表。它们被轮流传递给相关函数，视情况而定是 `online` 或是 `offline`。这两个函数对 `/sys/devices/system/node/node*/cpu${1}/online` 回显 0(离线)或 1(在线)。因为无论 CPU 是在哪个 NUMA 节点下，它都有一个唯一的 ID，所以*能使脚本免于计算出使用的是哪个节点。例如只有一个 `node*/cpu2`。在这之前，这些函数做了两项合理检查。第一项检查是查看该 CPU 是否存在一个 `online` 文件。在 x86 系统中，不可能使第一个 CPU 离线，而且第一个 CPU 也没有 `online` 文件。在这种情况下，显示消息 `CPU$i does not have online/offline functionality`，然后停止执行函数的余下部分。第二项检查是查看 CPU 是否已处于被请求状态。如果是，则显示一条(含义模糊的)消息 “`write error: Invalid argument`”。所以脚本最好将其隐藏起来，而是显示一条更有用的消息 `CPU$cpu is already Online`(根据情况可能是 `Offline`)。echo 语句之后，脚本测试返回码查看成功与否，并且向用户报告是否检测到错误。



在这样简短的函数中，使用嵌套 if 语句是可以接受的。在较长的函数中，第一个检测 `online` 文件是否存在的错误测试可以显式地从函数中返回，虽然使得函数的余下部分稍显不美观，但还是可以让读者清楚地看出函数在 `online` 文件不存在的情况下不再执行任何操作。

在这些运行示例中，本节之前的 `cpuinfo.sh` 脚本也可用来帮助识别每个系统中的可用 CPU。首先，对于便携式电脑，执行 `cup.sh` 产生的结果没有太多可言。因为无法关闭 CPU0，所以只能关闭或打开 CPU1。

```
laptop# ./cpu.sh
node0/cpu0 is always Online.
Usage:
  cpu.sh show [ cpu# ] - shows all CPUs shared by that node
  cpu.sh on cpu# ( cpu# cpu# cpu# ... )
  cpu.sh off cpu# ( cpu# cpu# cpu# ... )
laptop# ./cpuinfo.sh
laptop has 1 physical CPUs (Pentium(R) Dual-Core CPU T4400 @ 2.20GHz) each with 2 cores.
Each core has 2 threads: total 4 threads
laptop# ./cpu.sh show
node0/cpu0 is always Online.
node0/cpu1 is Online
laptop# ./cpu.sh off 1
node0/cpu0 is always Online.
node0/cpu1 is Online - Offlining CPU1 ... OK
laptop# ./cpu.sh show
node0/cpu0 is always Online.
node0/cpu1 is Offline
laptop# ./cpu.sh on 1
node0/cpu0 is always Online.
node0/cpu1 is Offline - Onlining CPU1 ... OK
laptop# ./cpu.sh show
```

```

node0/cpu0 is always Online.
node0/cpu1 is Online
laptop# ./cpu.sh off 0
node0/cpu0 is always Online.
CPU0 does not have online/offline functionality
laptop# ./cpu.sh show
node0/cpu0 is always Online.
node0/cpu1 is Online
laptop#

```

在更大的 8 核系统中，该脚本会显得更有趣。单个处理核会按照要求被启用或禁用。每个核依然是同一 NUMA 节点的一部分，这本质上说明了它完全是一个非 NUMA 架构。

```

minnow# ./cpuinfo.sh
minnow has 2 physical CPUs (Intel(R) Xeon(R) CPU L5420 @ 2.50GHz) each with 4 core
s. Each core has 2 threads: total 16 threads
minnow# ./cpu.sh show
node0/cpu0 is always Online.
node0/cpu1 is Online
node0/cpu2 is Online
node0/cpu3 is Online
node0/cpu4 is Online
node0/cpu5 is Online
node0/cpu6 is Online
node0/cpu7 is Online
minnow# ./cpu.sh off 2 5 7
node0/cpu0 is always Online.
Offlining CPU2 ... OK
Offlining CPU5 ... OK
Offlining CPU7 ... OK
minnow# ./cpu.sh show
node0/cpu0 is always Online.
node0/cpu1 is Online
node0/cpu2 is Offline
node0/cpu3 is Online
node0/cpu4 is Online
node0/cpu5 is Offline
node0/cpu6 is Online
node0/cpu7 is Offline
minnow#

```

如果不是总能立刻对这样的机器进行访问，则来自这一具有 128GB 内存、8 个 CPU(一共 48 个核)的 SunFire X4640 的输出可能比较具有参考价值。

```

whopper# ./cpuinfo.sh
whopper has 8 physical CPUs (Six-Core AMD Opteron(tm) Processor 8435) each with 6 c
ores. Each core has 2 threads: total 96 threads
whopper# ./cpu.sh show

```

```
node0/cpu0 is always Online.
node0/cpu1 is Online
node0/cpu2 is Online
node0/cpu3 is Online
node0/cpu4 is Online
node0/cpu5 is Online
node1/cpu10 is Online
node1/cpu11 is Online
node1/cpu6 is Online
node1/cpu7 is Online
node1/cpu8 is Online
node1/cpu9 is Online
node2/cpu12 is Online
node2/cpu13 is Online
node2/cpu14 is Online
node2/cpu15 is Online
node2/cpu16 is Online
node2/cpu17 is Online
node3/cpu18 is Online
node3/cpu19 is Online
node3/cpu20 is Online
node3/cpu21 is Online
node3/cpu22 is Online
node3/cpu23 is Online
node4/cpu24 is Online
node4/cpu25 is Online
node4/cpu26 is Online
node4/cpu27 is Online
node4/cpu28 is Online
node4/cpu29 is Online
node5/cpu30 is Online
node5/cpu31 is Online
node5/cpu32 is Online
node5/cpu33 is Online
node5/cpu34 is Online
node5/cpu35 is Online
node6/cpu36 is Online
node6/cpu37 is Online
node6/cpu38 is Online
node6/cpu39 is Online
node6/cpu40 is Online
node6/cpu41 is Online
node7/cpu42 is Online
node7/cpu43 is Online
node7/cpu44 is Online
node7/cpu45 is Online
node7/cpu46 is Online
node7/cpu47 is Online
whopper# ./cpu.sh off 0 3 7 9 12 24 31 42 45
node0/cpu0 is always Online.
```

```
CPU0 does not have online/offline functionality
Offlining CPU3 ... OK
Offlining CPU7 ... OK
Offlining CPU9 ... OK
Offlining CPU12 ... OK
Offlining CPU24 ... OK
Offlining CPU31 ... OK
Offlining CPU42 ... OK
Offlining CPU45 ... OK
whopper# ./cpu.sh show
node0/cpu0 is always Online.
node0/cpu1 is Online
node0/cpu2 is Online
node0/cpu3 is Offline
node0/cpu4 is Online
node0/cpu5 is Online
node1/cpu10 is Online
node1/cpu11 is Online
node1/cpu6 is Online
node1/cpu7 is Offline
node1/cpu8 is Online
node1/cpu9 is Offline
node2/cpu12 is Offline
node2/cpu13 is Online
node2/cpu14 is Online
node2/cpu15 is Online
node2/cpu16 is Online
node2/cpu17 is Online
node3/cpu18 is Online
node3/cpu19 is Online
node3/cpu20 is Online
node3/cpu21 is Online
node3/cpu22 is Online
node3/cpu23 is Online
node4/cpu24 is Offline
node4/cpu25 is Online
node4/cpu26 is Online
node4/cpu27 is Online
node4/cpu28 is Online
node4/cpu29 is Online
node5/cpu30 is Online
node5/cpu31 is Offline
node5/cpu32 is Online
node5/cpu33 is Online
node5/cpu34 is Online
node5/cpu35 is Online
node6/cpu36 is Online
node6/cpu37 is Online
node6/cpu38 is Online
node6/cpu39 is Online
```

```
node6/cpu40 is Online
node6/cpu41 is Online
node7/cpu42 is Offline
node7/cpu43 is Online
node7/cpu44 is Online
node7/cpu45 is Offline
node7/cpu46 is Online
node7/cpu47 is Online
```

10.9.6 /sys/devices/system/node

`mem.sh` 脚本检查每个 CPU 节点的可用内存。在真实的 NUMA 系统中，如果一个节点的所有 CPU 都处于离线状态，则与该节点相关的内存不可用。尽管对于处理任务 CPU 被标记为离线状态，但按照 Linux 在 x86_64 架构下的运行方式，内存依然是可用的。



可从
wrox.com
下载源代码

```
$ cat mem.sh
#!/bin/bash

kb=`head -1 /proc/meminfo | awk '{ print $2 }'`
mb=`echo "scale=2; $kb / 1024" | bc`
gb=`echo "scale=2; $mb / 1024" | bc`
echo "Server has $gb Gb ($kb Kb)"

cd /sys/devices/system/node
grep MemTotal node/meminfo | while read name node memtotal kb kB
do
    mb=`echo "scale=2; $kb / 1024" | bc`
    gb=`echo "scale=2; $mb / 1024" | bc`
    echo "Node $node has $gb Gb. "\
    "`grep -w 1 /sys/devices/system/node/node${node}/cpu[0-9]*/online \
    | wc -l` CPUs online"
done
```

mem.sh

该脚本开始时显示 OS 中全部可用的 RAM。大小可能比实际安装的总 RAM 要稍小，因为其中一些内存存在启动时被 PCI 卡与其他系统设备进行了映射。随后，脚本从 `/sys/devices/system/node/node*/meminfo` 中读取 `MemTotal` 一行。对于返回的每一项（在 `meminfo:Node 0 MemTotal:3074028 kB` 一行显示一些信息），脚本读取变量 `name`、`node`、`memtotal`、`kb` 与 `kB`。不是所有这些都是脚本所需的。实际上，只有 `name` 与 `kb` 才被用到，但其他变量有助于在不借助 `awk` 的前提下对输出进行分析。

最后，脚本使用 `grep` 在与节点相关的每个 CPU 的 `online` 文件中搜索 `1` 的数目。这样就能知道该节点的在线 CPU 数目。这实际上忽略了 CPU0 不具有 `online` 文件这一事实。有一些变通方案，但该脚本的目的是说明 `/sys` 的工作方式，而不要用太多的错误检测来将脚本堆砌得混乱不堪。

```
whopper# ./mem.sh
Server has 125.91 Gb (132035676 Kb)
Node 0 has 15.76 Gb. 4 CPUs online
Node 1 has 15.78 Gb. 4 CPUs online
Node 2 has 15.78 Gb. 5 CPUs online
Node 3 has 15.78 Gb. 6 CPUs online
Node 4 has 15.78 Gb. 5 CPUs online
Node 5 has 15.78 Gb. 5 CPUs online
Node 6 has 15.78 Gb. 6 CPUs online
Node 7 has 15.78 Gb. 4 CPUs online
whopper# ./cpu.sh on `seq 0 50`
node0/cpu0 is always Online.
CPU0 does not have online/offline functionality
CPU1 is already Online
CPU2 is already Online
Onlining CPU3 ... OK
CPU4 is already Online
CPU5 is already Online
CPU6 is already Online
Onlining CPU7 ... OK
CPU8 is already Online
Onlining CPU9 ... OK
CPU10 is already Online
CPU11 is already Online
Onlining CPU12 ... OK
CPU13 is already Online
CPU14 is already Online
CPU15 is already Online
CPU16 is already Online
CPU17 is already Online
CPU18 is already Online
CPU19 is already Online
CPU20 is already Online
CPU21 is already Online
CPU22 is already Online
CPU23 is already Online
Onlining CPU24 ... OK
CPU25 is already Online
CPU26 is already Online
CPU27 is already Online
CPU28 is already Online
CPU29 is already Online
CPU30 is already Online
Onlining CPU31 ... OK
CPU32 is already Online
CPU33 is already Online
CPU34 is already Online
CPU35 is already Online
CPU36 is already Online
CPU37 is already Online
```

```
CPU38 is already Online
CPU39 is already Online
CPU40 is already Online
CPU41 is already Online
Onlining CPU42 ... OK
CPU43 is already Online
CPU44 is already Online
Onlining CPU45 ... OK
CPU46 is already Online
CPU47 is already Online
CPU48 does not have online/offline functionality
CPU49 does not have online/offline functionality
CPU50 does not have online/offline functionality
whopper#
```

10.9.7 sysctl

`sysctl` 命令能控制这些参数中的大多数。尽管对 `/proc` 回显适当的值可以随时对 Linux 系统进行重新配置，并且重启也会对 `/etc/sysctl.conf` 进行修改，但 `sysctl -p` 还是会动态地重新读取 `/etc/sysctl.conf` 的全部内容。对于仍然需要对内核修改进行重启的早期 Unix 而言，这是极其有利的。

10.10 本章小结

内核管理正在运行的系统的所有细节。进程是系统的一部分，而且从很早开始就在 `/proc` 虚拟文件系统中包含了进程表。Linux 内核在 `/proc` 中还包含了很多其他内核状态。这样一来使得文件系统更加灵活。大的内存页、共享内存与其他参数可以无须重启随时进行配置。

进程控制包括对进程的文件描述符的管理。文件描述符可以按照各种方式进行重定向来实现不同的功能。输入与输出可以通过管道连接，并在其他文件与进程之间重定向。

本章对进程进行了非常深入的介绍。之后的第 11 章将讨论另一些可用的 shell、它们的共同点、各自提供的特性以及相互之间的区别，并结束本书的第 I 部分。

第 11 章

shell 的选择与使用

在传统的 Unix 系统中，所有用户与系统脚本的标准 shell(/bin/sh)都是 Bourne shell。在大多数 GNU/Linux 发行版中，bash 总是默认的 shell。最近以来，一些 GNU/Linux 系统使用 dash 作为系统 shell(/bin/sh)，但保留 bash 作为交互式 shell，也就是/usr/bin/bash。有很多不同的 shell，它们编写出来都有独特的目的且有各自的历史。这使得它们的语法与特性集都存在一些区别。

本章介绍一些可供使用的 shell，以及它们的来源与用途。本书的重点在于 bash，对于 Bourne shell 只是提及一下，但了解其他一些流行的 shell 也很重要。本章同样希望能介绍一些不同的运行 shell 的环境，以及这些环境如何影响脚本的操作。

对于交互使用与 shell 脚本编程确实没有必要使用相同的 shell。使用相同 shell 的主要好处在于很容易对语法进行测试，或者在编写脚本之前可以交互式地试验某一想法来测试其运行情况。另一方面，使用具有文件系统导航、历史调用等特性的交互式 shell 可能用处更大，而且总是可以在命令行中键入 shell 名称来调用另一个 shell。

11.1 Bourne shell

在 Unix 诞生初期，它具有一个由 Unix 发明人之一 Ken Thompson 编写的基本 shell。Steve Bourne 于 1979 年编写了可编写脚本的 Unix shell——Bourne shell。所有其他 shell 都具有一个标志性前缀——ksh、csh、zsh 等——但 Bourne shell 不叫 bsh，因为它就是一个 shell，所以其常规路径为/bin/sh。随后产生了其他的 shell，并且具有更多特性，同时在总体上保持了与 Bourne shell 的兼容性——一些 shell 比其他 shell 兼容性更好。

Bourne shell 中最重要的新概念是管道。管道允许进程将其输出传递给另一个进程。这使得 shell 命令的功能发生了剧烈的变化。Bourne 还引入了变量与流控制，将 shell 从一个基本的命令解释器转变为灵活的脚本语言。

11.2 Kornshell

David Korn 于 1983 年编写了 Kornshell(ksh)。它是一个用于编写脚本与交互式使用的流行 shell，尤其是在专有 Unix 上。与 bash 和 dash 类似，它向后兼容 Bourne shell，但增加了新特性与新语法。ksh 引入了 shell 历史的光标键导航，还提供了数组与浮点数学。ksh 在很长一段时间内都是 AT&T 的专有 Unix 软件，所以 pdksh(现在的 mksh，参见 <http://mirbsd.de/mksh>)是 ksh93 的自由软件版本。在 ksh93 于 2005 年以 IBM 的通用许可协议发布后，大多数 GNU/Linux 发行版开始包含 ksh93 而不再是 pdksh 或 mksh，如 OpenSolaris。因此，只要在较新的系统中找到 ksh，则它很可能是真正的 ksh93，而不是一个克隆版本。

ksh 与 Bourne 功能的共同点被用来为 /bin/sh 定义 POSIX 标准，所以 ksh 是非常重要的 shell 脚本语言。在传统的 Unix 系统中，将 root 用户的 shell 设置为 /bin/ksh 是完全被认可的。它是 IBM 的 AIX Unix 的默认 shell。/etc/init.d 脚本仍然会在 Bourne shell 下运行，但交互式 root shell 可以是 ksh(通常使用 -o vi 选项来提供类似 vi 的历史回调)。

Microsoft 的 Services For Unix(SFU，目前已废止)为 Windows 环境提供了一个几乎兼容的 ksh shell，尽管 SFU 基于当时与原始 ksh 不是很兼容的 mksh。在 <http://lists.blu.org/pipermail/discuss/1998-August/002393.html> 中可以阅读到一段故事，是关于 David Korn 如何质疑 Microsoft 的产品经理在演示 SFU 时阐述的对于 Kornshell 实现的选择。Korn 批评了所选择的实现方案，因为该方案与真正的 ksh 不兼容，并且询问 Microsoft 是否考虑了任何更兼容的 ksh 变体。而直到可怜的 Microsoft 代表试图声明它们的 Kornshell 实现与 Kornshell 完全兼容之后，才向这位代表表明提出关于 Kornshell 兼容性的尴尬问题的是 David Korn 本人。

11.3 C shell

Bill Joy 于 20 世纪 70 年代编写了 C shell(csh)。Bill Joy 是 Sun Microsystems 的创始人之一，而且还是一位非常多产的 BSD Unix 黑客。csh 的主要吸引力之一是它的语法与很多系统程序员非常熟悉的 C 语言非常相似。它是首个提供 history 命令的 shell，并且是比 Bourne shell 更好的交互式 shell。csh 还增添了任务控制与用波浪号(~)代表当前用户的主目录的概念。所有这些特性(不包括 C 风格语法)都被这里列出的所有其他 shell 所采纳。

1996 年，Tom Christiansen 写了一篇被广泛发布的题为“Csh Programming Considered Harmful”的文章(<http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/>)。该文章指出 csh 语法的一些工作方式可能有悖直观，或者会限制系统程序员。Christiansen 提出的问题特别集中在重定向与进程控制方面。

11.4 Tenex C shell

tcsh 就是 Tenex Csh，它对标准的 csh 提供了很多改进之处，并保持对 csh 的完全兼容。

这些改进包括更好的历史控制、用于对目录位置进行堆栈操作的 `pushd` 与 `popd`、终端锁定、`which` 与 `where` 命令，以及只读变量。`tcsh` 还提供了拼写纠正功能。如果 `tcsh` 怀疑有键入错误，则会使用推荐选项来提示用户。

除了自动补全命令与文件名，`tcsh` 还增加了变量名的自动补全功能。可以设置使这一功能在区分大小写与不区分大小写两种模式下使用。

11.5 Z shell

Paul Falstad 于 1990 年编写了 Z shell(`zsh`)。原本是要编写一个类似 `ksh` 的 shell，但还包括了一些类似 `csch` 的特性，因为 `csch` 在 20 世纪 70 年代与 80 年代是非常流行的交互式 shell。将 Z shell 作为交互式 shell 是特别有益的。尽管 Z shell 的目标是对 `ksh` 兼容，但它不保证完全的 POSIX 或 Bourne 兼容性(可以带来更大的灵活性，从而添加新的特性)。使用 `emulate` 命令可以改变它的行为，或者以 `/bin/sh` 或 `/bin/ksh` 调用来使行为更像这些 shell。

`zsh` 在交互式使用方面很像 `bash`，并且提供了类似的历史回调与命令补全功能，但在某些方面还要更强大一些。`compctl` 命令可用来对补全方式进行自定义。`zsh` 的通配语法与 `ksh` 和 Bourne shell 有些不同，并且数组从 1 而不是 0 开始索引。

11.6 Bourne Again Shell

`bash` 是大多数 GNU/Linux 与 Mac OSX 系统的标准交互式 shell，并且在传统 Unix 用户中也开始流行起来。它也是 `cygwin` 环境(为 Microsoft Windows 环境下提供 GNU 工具)的默认 shell。它与 Bourne shell 兼容，但添加了一些额外特性，其中大部分都在本书中有介绍。`bash` shell 的名称(Bourne Again Shell)是对 Bourne shell 的作者的名字开的一个玩笑。

`bash` 最初由 Brian Fox 于 1988 年为自由软件基金会(Free Software Foundation, FSF)编写，现在由 Chet Ramey 来维护。它吸收了包括 `csch` 与 `ksh` 在内的各种 shell 的思想。最明显的是，`bash` 使用 `ksh` 中的 `[[...]]`、`$(...)` 与 `((...))` 语法。

如果以 `sh` 调用，则 `bash` 在其读取的配置文件中表现得更像 Bourne shell。本章后面会有更加详细的介绍。

11.7 Debian Almquist Shell

`dash` 以 Almquist Shell(`ash`)的形式于 1989 年诞生，作者是 Kenneth Almquist。1999 年，它以 Debian Almquist Shell(`dash`)的形式被 Herbert Xu 移植到 Debian 项目中。与 `bash` 的类似之处是以 POSIX 兼容性为目标，但不同的是，`dash` 没有做进一步的工作。它的目的仅仅是成为一个 POSIX 兼容的 shell。这使得它比 `bash` 更小、更轻量且更快。因此，它在很多 GNU/Linux 发行版(一般保留 `bash` 用于交互式使用)中取代 `bash` 作为默认的 `/bin/sh` 来用于系统脚本，尤其是启动脚本。

GNU/Linux 中长期作为/bin/sh 来用的 bash 在向 dash 迁移时导致了一些问题，因为将 /bin/sh 作为解释器的很多系统脚本希望能使用 bash 特性。<https://bugs.launchpad.net/ubuntu/+source/dash/+bug/61463> 站点给出了一个问题列表，它们是 Ubuntu 6.10 在 2006 年将 bash 向 dash 迁移以作为默认/bin/sh 时所遇到的。

11.8 点文件

操作系统以其标准格式提供各种各样的设置。这些设置可供发行商与项目进行选择以用于它们各自的 shell。系统管理员通过编辑/etc 目录中的配置文件可以对这些设置进行自定义。与所有的 Bourne 兼容的 shell 都相关的主要配置文件是/etc/profile。该文件为交互式 shell 提供了一些基本的合理设置。它通常按照是否以 root 运行来将命令提示符设置为\$ 或#。它还会设置 umask、PATH、TERM 以及其他有用变量。该文件还执行其他一些有用的任务，如显示/etc/motd(Message of the Day)文件，提示用户是否有新的邮件消息，甚至在用户登录时显示一条 fortune cookie 消息来逗逗用户。它也可以调用其他脚本，被调用脚本通常在/etc/profile.d/目录中。该目录中的这些文件可以用来对特定的 shell 特性、应用程序以及工具进行自定义，使得包管理器可以随着应用程序添加或删除这些文件，而不必对/etc/profile 脚本本身进行编辑。



所有这些配置文件都被 source，而不是简单地执行。也就是说，这些文件中定义的任何变量或函数被继承到运行 shell 中。

因为每个用户都希望按照各自的方式对各自的 shell 进行自定义，所以每个用户都有各自的一组文件。这些文件与全局的/etc 中的文件不同，用户可以按照各自的方式编辑它们。为了系统的稳定性，首先分析/etc 中的文件，随后是用户各自的文件。每个 shell 都有一套自己要读取的文件名，所以与 bash 相关的命令放置到~/.bashrc 中。如果用户在某些时候选择运行 ksh，这不会导致任何问题，因为 ksh 不会读取~/.bashrc 而是读取~/.kshrc。无论各自有什么约定，所有的 Bourne 兼容的 shell 都会读取/etc/profile 与~/.profile。类似地，基于 csh 的 tcsh 会读取 csh 的/etc/.login 与~/.cshrc 来保持对 csh 的兼容。

名称以点号开头的文件无法被常规的 ls 命令显示。这并不是什么安全措施——ls -a 可以很容易将它们显示出来——但有利于将配置文件与数据文件分开。主目录通常都会杂乱无章，所以让用户的自定义文件从默认的显示中隐藏起来会比较有用。所有这些配置文件(甚至是/etc 中的文件，其中大部分没有隐藏)经常被通称为“配置文件”或“环境”，也简称为“点文件”。

点文件带来了极大的可定制性与灵活性。但结果是一堆文件——/etc/profile、/etc/login、/etc/bash.bashrc、/etc/csh.cshrc、/etc/ksh.kshrc、/etc/profile.d/*、~/.profile、~/.bashrc、~/.kshrc、~/.login、~/.cshrc 与~/.tcshrc 等——导致很多 shell 用户与系统管理员感到更加困惑。绕口令 She sells sea-shells on the sea shore 的困难程度显然与所有这些文件的不同组合形式无法

比拟。何时读取哪个文件，什么是每个文件都要读取的？3 种不同的 shell 调用分别是交互式登录 shell、交互式非登录 shell 与非交互式 shell。每个都有各自的定义，并且每个 shell 根据自身情况读取不同的文件。随后几节将介绍所有这些组合方式。这些组合按照调用类型与 shell 来进行组织。



因为 dash 在这 3 种情况下与 Bourne shell(sh)表现一样，所以表中省略了它。

11.8.1 交互式登录 shell

系统登录事件的结果就是执行登录 shell，它天生就是交互式的。su -(不仅仅是 su；su 只产生交互式 shell，而不是登录 shell)会产生登录 shell，ssh 会话、控制台以及串行终端登录会话也会产生登录 shell。对于交互式登录 shell，让配置脚本执行像 eval `ssh-agent` 之类的操作比较有用。配置脚本执行的一系列操作包括在登录会话(尽管在图形化环境下，视窗系统本身可能会配置成 ssh-agent 的客户端)中为所有的命令保留 ssh 私钥、定义别名等。

还有一种情况比较有用：对系统只有非 root 权限，且不希望将系统提供的 shell 作为交互式 shell 调用，可以根据自己的选择使用适当的点文件来代替系统提供的 shell。为确保 Bourne shell 被 bash 替换，下面这一小段位于 ~/.profile 末尾的代码保证无论如何都能找到 bash。

```
if [ -x /usr/bin/bash ]; then
    echo "Replacing $SHELL with bash"
    exec /usr/bin/bash
fi
```



Debian GNU/Linux 系统中的配置是，/etc/profile 调用/etc/bash.bashrc，且默认的 ~/.profile 调用 ~/.bashrc，这样交互式登录 shell 会选取交互式非登录 shell 的所有文件。

表 11-1 显示了每个 shell 如何配置交互式登录 shell。

表 11-1 交互式登录 shell 的配置文件

shell	读取的配置文件
bash	首先读取/etc/profile，然后是 ~/.bash_profile、~/.bash_login 与 ~/.profile，首先找到哪个就读取哪个
csh	首先读取/etc/csh.cshrc，然后是/etc/csh.login、/etc/.login、/etc/login.std 或/etc/cshrc(具体取决于操作系统)，最后读取 ~/.cshrc 与 ~/.login
sh	首先读取/etc/profile，然后是 ~/.profile
tcsh	与 csh 相同，但如果 ~/.tcshrc 存在，则代替 ~/.cshrc

(续表)

shell	读取的配置文件
ksh	/etc/profile、~/.profile、/etc/ksh.kshrc 与 ~/.kshrc
zsh	/etc/zsh/zshenv、\$ZDOTDIR/.zshenv、/etc/zsh/zprofile、\$ZDOTDIR/.zprofile、/etc/zshrc、\$ZDOTDIR/.zshrc、/etc/zsh/slogin 与 \$ZDOTDIR/.zlogin

1. csh 兼容性

csh 组合方式实际上比表 11-1 更复杂。详见 tcsh(1)手册页的 FILES 部分，其中包含了 NeXT、Solaris、ConvexOS、Stellix、Intel、A/UX、Cray 与 Irix 之间 csh 的详细区别，以及 tcsh 在组合方式上的区别。

2. zsh 兼容性

根据 zsh 的构建方式，/etc/zsh 实际上可能是/etc。Solaris 的 zsh 在构建时没有使用 enable_etcdir 配置选项，所以/etc/zsh 在 Solaris 中就是/etc。这种构建 zsh 的不同也发生在其他一些发行版中：Red Hat 使用/etc，Debian 使用/etc/zsh。另外，如果没有设置 ZDOTDIR，则使用\$HOME 作为默认值。自定义 ZDOTDIR 的位置是/etc/zsh/zshenv，如果其中没有进行设置，则在~/.zshenv 中。

11.8.2 交互式非登录 shell

非登录 shell 是生成给已登录用户使用的 shell，但它并不代表新的登录实例。这种 shell 发生在当我们在已有的交互式 shell 中直接键入 bash 时，或者针对 su 命令(而不是 su -)，以及在图形化会话中打开新的终端模拟器窗口或选项卡时。非登录 shell 很适合用来自定义 PATH 变量、设置 PS1 提示符等。表 11-2 给出了每个 shell 配置交互式非登录 shell 的方式。

表 11-2 交互式非登录 shell 的配置文件

shell	读取的配置文件
bash	~/.bashrc，如果以 sh 形式调用则读取与 sh 一样的配置文件
csh	/etc/csh.cshrc 与 ~/.cshrc
sh	如果设置了\$ENV 则读取该文件，否则不读取任何文件
tcsh	首先读取/etc/csh.cshrc，然后如果找到~/.tcshrc 则读取它，否则读取~/.cshrc
ksh	/etc/ksh.kshrc 与 ~/.kshrc
zsh	/etc/zsh/zshenv、\$ZDOTDIR/.zshenv、/etc/zshrc 与 \$ZDOTDIR/.zshrc

这里，Bourne shell 读取环境变量\$ENV(如果存在该变量)命名的文件。因此它与 bash 的~/.bashrc 文件相似，但要稍微灵活一些。灵活性在于用户能够为非登录 shell 会话选择读取任何文件。为了保持与 Bourne shell 的兼容性，如果 bash 通过 sh 调用，则也会读取设置过的\$ENV 变量。这对于假设/bin/sh 是 Bourne shell 且会读取\$ENV 文件的早期脚本

比较有用。

11.8.3 非交互式 shell

非交互式 shell 是与终端没有直接联系的 shell。shell 脚本会产生一个非交互式 shell 会话，如 cron 与 at 这样的工具。表 11-3 列出了每种 shell 配置非交互式 shell 的方式。注意，Bourne shell 及其兼容的 shell 在运行非交互式 shell 时不会分析任何系统或用户级文件。

表 11-3 非交互式 shell 的配置文件

shell	读取的配置文件
bash	如果设置了\$BASH_ENV 则读取其表示的文件，否则与调用 sh 一样
csh	/etc/csh.cshrc 与 ~/.cshrc
sh	不读取
tcsh	首先读取/etc/csh.cshrc，然后如果找到 ~/.tcshrc 则读取它，否则读取 ~/.cshrc
ksh	不读取
zsh	/etc/zsh/zshenv 与 \$ZDOTDIR/.zshenv

与 sh 为交互式非登录 shell 读取设置过的\$ENV 一样(如果 bash 以 sh 方式调用，则与 sh 一样)，bash 也会读取\$BASH_ENV，前提是存在\$BASH_ENV，并且是以 bash 方式调用。这允许 bash 为非交互式脚本包含一些在交互式脚本中不会包含的额外配置。

11.8.4 登出脚本

bash、zsh 与 csh 也提供了在交互式 shell(无论是登录的还是非登录的)退出时执行的脚本。这在会话的终止意味着窗口的消失(如 Microsoft Windows 中的 PuTTY 会话)或者需要为调用系统对终端设置进行一些修改的时候非常有用。

对于 zsh 而言，它退出的时候会顺序执行\$ZDOTDIR/.zlogout 与/etc/zsh/zlogout。对于 bash 而言，它退出的时候会调用 ~/.bash_logout 与/etc/bash_logout。对于 csh 与 tcsh 而言，退出的时候会读取 ~/.logout、/etc/.logout、/etc/logout 与/etc/csh.logout，或者适当的等价脚本。另外，tcsh(1)手册页的 FILES 部分对各种不同操作系统的全局登出脚本文件名的使用约定进行了很好的归纳。

11.9 命令提示符

shell 有 4 种不同的命令提示符，分别是 PS1、PS2、PS3 与 PS4。PS 表示 Prompt String(提示字符串)。PS1 几乎总是被自定义，其他的几乎从不自定义。

11.9.1 PS1 提示符

PS1 在第 2 章中曾介绍过。它是位于每行开头的标准提示符。在本书的例子中，一般都会显示为一个美元符号与紧随其后的一个空格(“\$ ”)。这是在 Bourne 与 bash shell 中

表示非特权用户的标准方式。root 用户的 PS1 提示符按照约定设置为 “# ”，或者至少以它结尾。PS1 可以包含文本，但大多数 shell 也会在 PS1 变量中扩展各种特殊字符。在 bash 中，默认的非特权提示符是\s-\v\$，表示 shell 的名称(s)与版本(v)——如 “bash-4.1\$ ”。最有用的一些字符是\u(用户名)、\h(主机名)、\t(当前时间)与\d(日期)。

完整的 PS1 特殊字符集在 bash(1)手册页的 PROMPTING 部分有描述。可以使用一个更有意思的特殊字符，通过输入标准\0xx 形式的八进制字符来设置彩色的提示符。下面的 PS1 将提示符设置为红色：

```
PS1="\033[1;31m\u@\h\w\$ \033[0m "
```

刚开始\033[1;31m 中的数字 31 是表示红色的代码。 \033[0m 中最后的 0 表示灰色，所以它将输入设置为常规颜色。只有提示符本身是红色的。加 10 表示背景颜色，所以要使用红色背景就使用 41，而不是 31。

表 11-4 与表 11-5 列出了每种颜色与风格的编号。可以任意对它们进行组合以得到需要的效果。

表 11-4 提示符颜色(y)

编号(八进制)	颜 色
30	黑色
31	红色
32	绿色
33	黄色
34	蓝色
35	紫红色
36	蓝绿色
37	灰色
38	默认

表 11-5 外观设置(x)

编号(八进制)	外 观
0	黑暗
1	明亮
4	下划线
5	闪烁
7	反白

这些长长的字符串非常简单。 \033[x;ym 用 y 定义颜色，用 x 定义外观。如果 x 为 0，

则输出是暗色的, 1 则很明亮。4 会对文本加下划线, 5 则使文本闪烁(如果终端支持闪烁), 而 7 会使用默认的反白显示。我们可以使用完整的序列来组合这些选项。灰色文本为 0;37, 红色背景为 1;41(红色为 31+10), 所以我们可以将两者结合起来:

```
PS1="\033[0;37m\033[1;41m\u@\h:\w$ "
```



尽管 shell 在一行的末尾对文本的换行处理很到位, 但当引入了上面那些控制字符后, 可能将换行处理搞砸。因此, 当到达窗口右边时, 文本不会向下滚屏, 而是自行换行。

11.9.2 PS2、PS3 和 PS4 提示符

PS2 是辅助提示符。它主要用在交互式编写的循环中, 如 while 循环。其默认值是>, 且一般不应当对其进行修改。

PS3 变量用在内置循环 select 中。select 在第 6 章介绍过。该变量只需要在使用内置命令 select 时设置就行。它能用来创建一个普通的菜单系统, 而不用功能多么完整或外观多么吸引人。

当 shell 使用 -x 选项(用于跟踪脚本的执行)时, 显示的是 PS4 变量。默认值是+, 而且与 PS2 类似的是, 没有必要修改。下面的脚本说明了如何使用 PS4 提示符的数目来显示间接调用的层数。脚本本身会显示一个加号(+), 且外部命令(expr)会显示两个加号(++)。进一步的命令(如果对他们再次使用了 -x 选项——该选项的继承不会超过第一代)可能具有 3 个加号(+++)的前缀。

```
$ cat while.sh
#!/bin/bash

i=1
while [ "$i" -lt "100" ]
do
    echo "i is $i"
    i=`expr $i \* 2`
done
echo "Finished because i is now $i"
$ bash -x while.sh
+ i=1
+ '[' 1 -lt 100 ']'
+ echo 'i is 1'
i is 1
++ expr 1 '*' 2
+ i=2
+ '[' 2 -lt 100 ']'
+ echo 'i is 2'
i is 2
```



```
++ expr 2 '*' 2
+ i=4
+ '[' 4 -lt 100 ']'
+ echo 'i is 4'
i is 4
++ expr 4 '*' 2
+ i=8
+ '[' 8 -lt 100 ']'
+ echo 'i is 8'
i is 8
++ expr 8 '*' 2
+ i=16
+ '[' 16 -lt 100 ']'
+ echo 'i is 16'
i is 16
++ expr 16 '*' 2
+ i=32
+ '[' 32 -lt 100 ']'
+ echo 'i is 32'
i is 32
++ expr 32 '*' 2
+ i=64
+ '[' 64 -lt 100 ']'
+ echo 'i is 64'
i is 64
++ expr 64 '*' 2
+ i=128
+ '[' 128 -lt 100 ']'
+ echo 'Finished because i is now 128'
Finished because i is now 128
$
```

11.10 别名

别名是交互式 shell 特有的。它们在 shell 脚本中不会扩展。这可以确保脚本的行为与预期一致，也意味着可以放心地为交互式使用定义别名，因为我们知道它不会对 shell 脚本造成破坏。这样一来，别名成为了简写的理想方式，可以避免不断重复地键入同一个复杂的单词或命令序列，也能让我们自动调整常用命令的行为。

11.10.1 节省时间

一些节省时间的别名包括 ssh 到特定主机、编辑特定文件等的快捷方式。较为便利的节省时间的方法是为经常登录的系统设置别名。有了后面的别名，我们可以以 apache 用户的身份登录到 Web 服务器，并以绑定用户的身份登录每个 DNS 服务器，而这些只需要输入 web、dns1 或 dns2。

```
$ alias web='ssh apache@web.example.com'
$ alias dns1='ssh bind@ns1.example.com'
$ alias dns2='ssh bind@ns2.example.com'
```

可以更进一步地添加更多选项。下面的别名模拟了一个简单的 VPN，它将局域网 Web 服务器的 80 端口转发到本机的 8080 端口。关于 ssh 端口转发，详见 ssh 文档。

```
$ alias vpn='ssh -q -L 8080:192.168.1.1:80 steve@intranet.example.com'
$ vpn
steve@intranet.example.com's password:<enter password>
Linux intranet 2.6.26-2-amd64 #1 SMP Sun Jun 20 20:16:30 UTC 2010 x86_64
You have new mail.
Last login: Wed Mar 16 21:45:17 2011 from 78.145.17.30
steve@intranet:~$
```

11.10.2 修改行为

使用别名来修改命令的标准行为是非常有用的做法。一个较为便利的尤其适合命令行新手的别名是 `alias rm='rm -i'`。它会强制 `rm` 命令在删除任何文件之前都有提示。可能会有争议说，这破坏了“除非真的有错误，否则就让所有操作都静默执行”这一 Unix 模型，但这对一些用户而言的确是有用的。如第 2 章提到的，另一个有用的别名是 `less -X`，它防止 `less` 在退出的时候刷新整个屏幕。只要我们发现自己经常性地输入某些序列，就应当停下来思考是否可以使用别名使操作更方便。`rm` 别名总是可以通过 `rm -f` 格式重写。我们还可以通过使用前缀反斜线来防止别名扩展，如下所示：

```
$ alias echo='echo Steve Says: '
$ echo Hello World
Steve Says: Hello World
$ \echo Hello World
Hello World
$
```

另一个有用的别名是 `wget` 的别名。该命令可能有很多参数，它们可以在 `~/.wgetrc` 中，但作为别名或许更为方便。一些 Web 服务器试图根据请求页面的浏览器来对内容进行自定义，所以向服务器发送比 `Wget/1.12(linux-gnu)` 更有用的用户代理字符串能得到更好的结果。`Mozilla/5.0` 就是一个比较有用的字符串。大多数标准的 Web 浏览器都会在用户代理字符串中包含 `Mozilla`。

```
$ alias download='wget -U"Mozilla/5.0" -nc '
$ download http://www.example.com/
--2011-03-16 21:52:06-- http://www.example.com/
Resolving www.example.com... 192.0.32.10, 2620:0:2d0:200::10
Connecting to www.example.com|192.0.32.10|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://www.iana.org/domains/example/ [following]
```

```
--2011-03-16 21:52:07-- http://www.iana.org/domains/example/
Resolving www.iana.org... 192.0.32.8, 2620:0:2d0:200::8
Connecting to www.iana.org|192.0.32.8|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2945 (2.9K) [text/html]
Saving to: `index.html'

100%[=====>] 2,945      --.-K/s  in 0s

2011-03-16 21:52:07 (83.8 MB/s) - `index.html' saved [2945/2945]

$ ls -l index.html
-rw-r--r-- 1 steve steve 2945 2011-02-09 17:13 index.html
$
```



一些网站使用 Google 搜索引擎来分享其全部内容，从而获得较高的排名。但当我们试图访问这些站点时，它们要求注册并登录。我们可以通过将用户代理字符串设置为 Googlebot/2.1(+http://www.google.com/bot.html)来骗过这些站点。

11.11 history 命令

history 命令列出已经运行过的命令，无论这些命令是在当前 **shell** 会话还是其他会话(必须是由同一个用户运行)中。该命令一般会将这些信息存储在用户主目录的<shell 名称>_history 文件中。这可以用来对运行过的命令进行审核与检查，但它并不是一个安全特性。历史文件是纯文本文件，并且很容易被任何有访问权限的人编辑。

11.11.1 回调命令

我们可以用多种方式进行命令回调；最基本的标准方法是运行 **history** 命令。它显示存储在当前历史缓冲区中的命令，每个命令左边还带有一个编号：

```
$ history | tail -7
557 pwd
558 cd
559 cd bin
560 ls
561 cat dos2unix
562 id
563 history
$
```

这些命令可以用 **bang**、**pling** 或者感叹号(!)进行回调，所以!562 会重新执行 **id** 程序。另外，!-n 回调倒数第 n 个命令。所以从上面 **shell** 运行所至的位置开始，!-1(或者简写!!)会再次调用 **history**，!-2 会调用 **id**，而!-3 则会调用 **cat dos2unix**。

11.11.2 搜索历史

主要有 3 种交互式搜索历史缓冲区的方式。第一种是使用箭头键。按上箭头向后滚动，按下箭头向前滚动。当重新得到需要执行的命令后，要么像以前一样按回车键来执行，要么使用左右箭头键来编辑命令，然后重新执行。

另外，如果准确地知道命令的首字母，则可以键入 `!` 与命令首字母来回调所需命令。如果要在命令行末尾追加额外文本，则将文本输入到需要调用的命令之后。这只对命令起作用，不包括命令的参数。

```
$ echo Hello World
Hello World
$ echo Goodbye
Goodbye
$ !e ←————— !e 回调最后一个以字母 e 开头的命令。
echo Goodbye
Goodbye
$ !echo ←————— !echo 回调最后一个 echo 命令。
echo Goodbye
Goodbye
$ !echo H ←————— !echo H 不会回调 echo hello world 命令，
echo Goodbye H      只是将 H 追加到最后一个 echo 命令之后。
Goodbye H
$
```

第三种搜索历史的方式(会搜索整个命令行)是按下 `Ctrl` 与 `r(^R)`，然后输入所要搜索的命令行的一部分。这样会向后搜索命令行历史，并找到最近的包含输入文本的一行命令。从这一点开始，再次按下 `^R` 可以得到次最近的匹配行。与使用箭头键浏览一样，我们可以按左右箭头键来编辑选择的命令行，然后执行。

11.11.3 时间戳

如果设置了 `HISTTIMEFORMAT` 变量，`bash` 会在历史文件中为命令行保存时间戳。然后当运行 `history` 时，命令会包含由 `HISTTIMEFORMAT` 变量指定格式的时间。该变量值定义在 `strftime(3)` 手册页中。`%c` 使用当前区域设置首选的格式，`%D` 是 `%m/%d/%y`(月/日/年)的缩写，`%F` 表示 `%Y-%m-%d`，`%T` 表示 `%H:%M:%S`(时:分:秒)。还有更多格式(完整列表见 `strftime(3)` 手册页)，但这些都是 `HISTTIMEFORMAT` 的最常用格式。

如果没有设置 `HISTTIMEFORMAT`，则不会保存时间戳。因为不能被更新，如果 `HISTTIMEFORMAT` 之前没有保存过，则之前的命令看起来像是与设置 `HISTTIMEFORMAT` 之前的最后一条命令同一时间运行的。在下面这个例子中，没有为 `root` 设置过 `HISTTIMEFORMAT`，所以在 21:19:19 这一时刻设置后，所有之前运行的命令都被打上了相同的时间戳，也就是当前 `shell` 启动的时间(21:19:08)，但这些命令运行的时刻要早于 `shell` 启动时刻。

```

# history
1 cd /boot/grub
2 vi menu.lst
3 cat menu.lst
4 date
5 ls
6 cat /etc/hosts
7 ssh steve@node4
8 vi /boot/grub/menu.lst
9 /sbin/reboot
10 PS1="# "
11 history
# date
Thu Mar 17 21:19:19 GMT 2011
# HISTTIMEFORMAT="%T %F "
# history
1 21:19:08 2011-03-17 cd /boot/grub
2 21:19:08 2011-03-17 vi menu.lst
3 21:19:08 2011-03-17 cat menu.lst
4 21:19:08 2011-03-17 date
5 21:19:08 2011-03-17 ls
6 21:19:08 2011-03-17 cat /etc/hosts
7 21:19:08 2011-03-17 ssh steve@node4
8 21:19:08 2011-03-17 vi /boot/grub/menu.lst
9 21:19:08 2011-03-17 /sbin/reboot
10 21:19:13 2011-03-17 PS1="# "
11 21:19:14 2011-03-17 history
12 21:19:19 2011-03-17 date
13 21:19:30 2011-03-17 HISTTIMEFORMAT="%T %F "
14 21:19:31 2011-03-17 history
#

```

当前 shell 的历史会被缓冲直到 shell 退出，所以我们不能看到历史文件被实时地更新。通过登出与再次登录，历史文件显示了带有时间戳的命令(以一个井号与数字开头)与之前的没有时间戳的命令。

```

# cat .bash_history
cd /boot/grub
vi menu.lst
cat menu.lst
date
ls
cat /etc/hosts
ssh steve@node4
vi /boot/grub/menu.lst
/sbin/reboot
#1300396753 ←————— 时间戳从这里开始
PS1="# "
#1300396754

```

```

history
#1300396759
date
#1300396770
HISTTIMEFORMAT="%T %F "
#1300396771
history
#1300396789
cat ~/.bash_history
#

```

11.12 Tab 补全

所有的现代 shell 都具有使用<Tab>键来补全命令与文件名的功能。Bourne shell 与 csh 没有该功能，但 ksh、bash、tcsh 与 zsh 都在不同程度上支持 Tab 补全。所有这些 shell 的基本原理都相同：输入单词的开头，然后按<Tab>键两次，shell 就会显示可能的命令与文件列表。具体细节因实现不同而有差异，所以下面将简要介绍这些 shell 中的补全方法。

11.12.1 ksh

输入 **ca**，然后按<TAB>键两次，得到一个类似 **select** 命令产生的列表。其中每一项都被编号，以便简单地输入与选择相关的编号，再加上一个<TAB>。例如，**ca** 可以是 **cancel**、**callgrind_annotate** 或者任何其他可能单词的开头。

```

ksh$ ca<TAB><TAB>
1) /usr/bin/cancel
2) /usr/bin/callgrind_annotate
3) /usr/bin/cameratopam
4) /usr/bin/callgrind_control
5) /usr/bin/cancel.cups
6) /usr/bin/cal
7) /usr/bin/captaininfo
8) /usr/bin/catchsegv
9) /usr/bin/card
10) /usr/sbin/cacertdir_rehash
11) /usr/sbin/callback
12) /bin/cat
6<TAB>
ksh$ /usr/bin/cal
      March 2011
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
$

```

文件名补全与命令行补全完全一样：

```
ksh$ cat /etc/host<TAB><TAB>
1) host.conf
2) hosts
3) hosts.allow
4) hosts.deny
4<TAB>
ksh$ cat /etc/hosts.deny
```

11.12.2 tcsh

tcsh 的补全方式基本上与 **ksh** 一样，但选项是以更有效的列式格式显示的。另外，没有必要按<TAB>键两次。对于 **tcsh**，按一次<TAB>键就够了。文件名补全与命令行补全完全一样。

```
goldie:~> ca<TAB>
cabextract      callgrind_annotate  captainfo  case.sh
cal              callgrind_control   case        cat
calendar        canberra-gtk-play   casela.sh  catchsegv
calibrate_ppa    cancel              case2.sh   catman
goldie:~> cal
      March 2011
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

goldie:~>
```

11.12.3 zsh

zsh 中的显示与 **tcsh** 非常相似。当按下<TAB>键一次时，选项被列在光标下的几列中，但被分为几类：外部命令、保留字、**shell** 内置命令等。每次按下<TAB>键，当前命令行会轮流选中不同的命令，所以按下回车键就可以选择当前显示的选项。当选择了所需的命令时，选项列表会被隐藏起来，这样显示更清晰。

```
steve@goldie> ca<TAB>
Completing external command
Cabextract      calibrate_ppa      canberra-gtk-play  cat
cal              callgrind_annotate cancel              catchsegv
calendar        callgrind_control  captainfo           catman
Completing reserved word
Case
steve@goldie> cal
```

```

    March 2011
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

steve@goldie>

```

11.12.4 bash

bash 的 Tab 补全与其他 shell 的工作方式非常相似。再来看一下 cal 命令：尽管 bash 需要输入两次<TAB>，但输入 ca<TAB><TAB>的结果与之前的 tcsh 例子很相似。文件名补全与命令行补全相同：输入 cat /etc/ho<TAB>会扩展成/etc/host，然后输入两个<TAB>会列出所有可用的选项。应当尽量减少选项的数目。

```

$ ca<TAB><TAB>
cabextract      calibrate_ppa   captainfo       catchsegv
cal              caller          case            catman
calendar        cancel          cat

$ cal

    March 2011
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

$ cat /etc/ho<TAB>st<TAB>
host.conf      hostname        hosts           hosts.allow     hosts.deny
$ cat /etc/hosts

```

bash 的补全功能也可以通过 bash_completion 包来进行编程。/etc/bash_completion 与 /etc/bash_completion.d/ 已经包含了各种命令的脚本。例如，bash_completion 知道 ssh 选项，并且能自动对其进行扩展。下面的一段代码显示 ssh 选项被 bash shell 自动补全。这比起仅仅是文件名补全给人印象更加深刻。由于命令是 ssh，所以它按特定方式进行补全。对于 gzip，bash 的补全功能会提供适合于 gzip 的选项。

```

$ ssh -o F<TAB>orward<TAB>
ForwardAgent=      ForwardX11=      ForwardX11Trusted=
$ ssh -o ForwardX<TAB>11<TAB><TAB>
ForwardX11=        ForwardX11Trusted=
$ ssh -o ForwardX11=no user@example.com

```


11.13 后台、前台与作业控制

运行交互式命令非常有用，但要利用 OS 的多任务特性，要么为每个要运行的命令开启一个新的终端会话，要么进程必须执行时不独占当前终端。后者可以通过在后台运行任务来实现。这些命令会像正常情况一样运行，但开始运行后会立刻返回到 PS1 提示符。如果后台进程被终止，会得到关于其状态的提示信息；如果进程结束，则会得到退出码。

11.13.1 后台进程

在后台执行命令的方法是在命令所在行的最后添加&符号。shell 将显示作业 ID(在方括号内)与后台进程的 PID。另外，我们可以用\$!变量访问命令的进程 ID。这意味着可以在进程结束之前将其关闭，或者等待某个特定进程。

当按照默认方式不使用任何参数调用 shell 内置命令 wait 时，它会等待所有的后台进程结束。在这种情况下，当属于当前 shell 的所有后台进程完成后，wait 返回 0。如果传递给 wait 的进程 ID(PID)或作业号不存在，或者其父进程不是当前 shell，则 wait 会立刻以返回码 127 退出。另外，wait 的返回码与被等待的进程或作业的返回码相同。所以，交互式用户或者脚本可以开启一组后台任务，选择等待其中一个，然后休眠直到该任务结束。下面的例子演示了这种工作方式。通过将\$!值保存在变量\$one、\$two 与\$three 中，第二个后台任务可以被选择用于 kill、wait 或其他命令。

```
$ sleep 600 &
[1] 13651
$ one=$!
$ sleep 600 &
[2] 13652
$ two=$!
$ sleep 600 &
[3] 13653
$ three=$!
$ echo PIDs are $one $two $three
PIDs are 13651 13652 13653
$ ps -fp $two
UID      PID  PPID  C  STIME  TTY      TIME CMD
steve 13652 13639 0 21:20   pts/6  00:00:00 sleep 600
$ kill -9 $two
$
[2]-  Killed                  sleep 600
$
```

11.13.2 作业控制

还有一个更好的控制进程的方式。作业控制是比进程控制更高级别的抽象。每个后台进程(或者管道)都被赋予了下一个顺序可用的作业号。每个 shell 运行实例都有各自的作业列表，所以当前 shell 运行的第一个作业将总是%1，且第二个总是%2，依此类推。当所有

作业都结束时，下一个还是会被赋为%1。内置命令 `jobs` 会列出 shell 所有作业的作业号、状态与命令行。`jobs -l` 还会显示作业的 PID。作业号后面有一个+，表示 shell 可见的默认作业或者“当前”作业。当前作业可以通过简写%%、%或%+来引用。

`fg` 与 `bg` 命令分别将指定的作业(或者在不指定时表示当前作业)带到前台与后台。按下 `Ctrl` 与 `z(^Z)` 会停止当前的前台进程，并将用户返回到交互式 shell 提示符。前台作业是被绑定到当前终端用于输入与输出的作业，所以它能从键盘接受交互式输入以及向屏幕进行写操作。下面的交互式会话演示了 3 个同时压缩 3 个 CD 的 ISO 映像的 `gzip` 实例。操作可能花一些时间，所以有足够的时间修改与查看这 3 个后台作业的状态。`sleep` 命令对于这种类似的演示程序非常有用，但 `gzip` 是一个更加真实的例子。通过观察生成的 `.gz` 文件大小是否一直在增长，我们可以在压缩中途将进程停止和/或关闭，这样就能看到 `gzip` 的进度。

```
$ ls
OEL5-cd1.iso  OEL5-cd2.iso  OEL5-cd3.iso
$ gzip *cd1.iso & gzip *cd2.iso & gzip *cd3.iso &
[1] 3224
[2] 3225
[3] 3226
$ jobs -l
[1] 3224 Running      gzip *cd1.iso &
[2]- 3225 Running      gzip *cd2.iso &
[3]+ 3226 Running      gzip *cd3.iso &
$ fg %2 ← 将作业%2 提到前台 (fg)
gzip *cd2.iso
^Z ← 然后停止作业
[2]+ Stopped          gzip *cd2.iso
$ bg %2 ← 然后带到后台
[2]+ gzip *cd2.iso &
$ kill -9 %3
$
[3]+ Killed          gzip *cd3.iso
$ jobs -l
[1]- 3224 Running      gzip *cd1.iso &
[2]+ 3225 Running      gzip *cd2.iso &
$ kill -9 % ← 通过作业号关闭作业 3；然后关闭任何当前作业
[2]+ 3225 Killed      gzip *cd2.iso
$ jobs -l
[1]- 3224 Running      gzip *cd1.iso &
$
$ fg
gzip *cd1.iso
^Z
[1]+ Stopped          gzip *cd1.iso
$ bg
[1]+ gzip *cd1.iso &
$ wait ← 等待最后一个作业完成
[1]+ Done            gzip *cd1.iso
$ jobs -l
$
```

当需要交互式输入时，后台作业的状态可能从运行变为停止。随后，我们可能需要将作业提到前台并提供所需的输入。然后我们可以按下 **Ctrl** 与 **z(^Z)**再次将其停止，并使用 **bg** 命令将其再次带到后台。下面这个简短的脚本在询问用户名称前休眠了 3 秒钟。然后在响应之前又休眠了 3 秒钟。这是很多脚本与程序的简化方式，如第三方的软件安装例程，这些例程可能在一段较长时间的静默状态之后突然意料之外地变得具有交互性。

```
# cat slowinstaller.sh
#!/bin/bash

sleep 3
read -p "What is your name? " name
sleep 3
echo Hello $name
# ./slowinstaller.sh
What is your name? Steve
Hello Steve
#
```

当在后台运行时，如果作业要求交互式输入，则它会被停止。在下面的代码中，用户看到提示“**What is your name?**”并进行回答。然而由于脚本处于后台，所以答案(**Bethany**)直接发送给交互式 shell(结果是报告“**Bethany: command not found**”)。

```
# ./slowinstaller.sh &
[1] 10661
# What is your name? Bethany
-bash: Bethany: command not found

[1]+  Stopped                  ./slowinstaller.sh
```

该脚本已经停止，所以必须提到前台(**fg**)接受输入。此处，用户意识到所发生的情况，并将进程提到前台继续交互式会话。

```
# ./slowinstaller.sh &
[1] 10683
# What is your name? fg
./slowinstaller.sh
Bethany
Hello Bethany
#
```

如果用户意识到提示来自后台任务，则她会简单地按回车键来显示下一个提示符。在这一阶段，shell 也会显示后台作业改变后的状态。用户然后可以将作业提到前台，并与其交互直到结束，如下面代码所示。交互结束后，进程可以被安全地停止(^Z)，然后再带到后台，直到它又需要交互式输入。或者像下面代码中的那样，[1]+ Done 消息表示进程成功地运行完毕。

```
# ./slowinstaller.sh
^Z
[1]+  Stopped                  ./slowinstaller.sh
#
# fg
./slowinstaller.sh
What is your name? Emily
^Z
[1]+  Stopped                  ./slowinstaller.sh
#
# bg
[1]+  ./slowinstaller.sh &
# Hello Emily

[1]+  Done                      ./slowinstaller.sh
#
```

11.13.3 nohup 和 disown

按照标准, 当登录 shell 退出时, 会向所有子进程发送 HUP(挂起)信号。这导致所有的后台任务在它们的交互式登录 shell 终止时也都会终止。bash 可以使用 shell 选项 `huponexit` 避开这一点, 但事先处理这种情况的标准方式是使用 `nohup` 命令。`nohup` 确保任务在属主 shell 退出时不会终止, 这对长时间运行的任务特别有用。最好是能够连接到系统, 开始某个任务, 然后再立即断开连接。没有 `nohup` 的话, 则必须让交互式登录 shell 保持在活动状态。这意味着必须保持网络连接, 并且连接至的机器一直开启, 不掉电也不崩溃, 不以任何理由关闭交互式 shell。

```
$ nohup /usr/local/bin/makemirrors.sh > /var/tmp/mirror.log 2>&1 &
[1] 14322
$ cat /var/tmp/mirror.log
Wed Mar 16 22:27:31 GMT 2011: Starting to resync disk mirrors.
Do not interrupt this process.
$ exit
logout
Connection to node3 closed.
```

与 `nohup` 相关的是 `disown`。已经运行的命令可以按与 `nohup` 进程被自动 `disown` 一样的方式被 `disown`。

```
node1$ sleep 500 &
[1] 29342
node1$ disown %1
node1$ exit
logout
Connection to node1 closed.
node7:~$ ssh node1
steve@ node1's password:
```

```
Linux node1 2.6.26-2-amd64 #1 SMP Sun Jun 20 20:16:30 UTC 2010 x86_64
Last login: Wed Mar 16 22:30:50 2011 from 78.145.17.30
node1$ ps -eaf|grep sleep
steve      29342      1  0 22:30 ?        00:00:00 sleep 500
node1$
```

11.14 本章小结

Unix 与 Linux 系统中有很多不同的 shell。本章介绍了一些最流行的 shell。它们表面看起来都非常相似：提供到系统的交互式命令行接口，用相同的方式运行基本的 shell 脚本。但实际上它们之间是存在天壤之别的，这使得将脚本从一个 shell 移植到另一个 shell 非常耗时且繁琐。

理解系统之间的差异是编写具有可移植性与健壮性的 shell 脚本的关键，也有利于与使用不同配置方法的客户系统的兼容性。另外，使用别名、提示符与配置文件可以使用户更熟悉 home 系统并因此效率更高，而无论是位于本地网络、局域网或者是企业范围的。

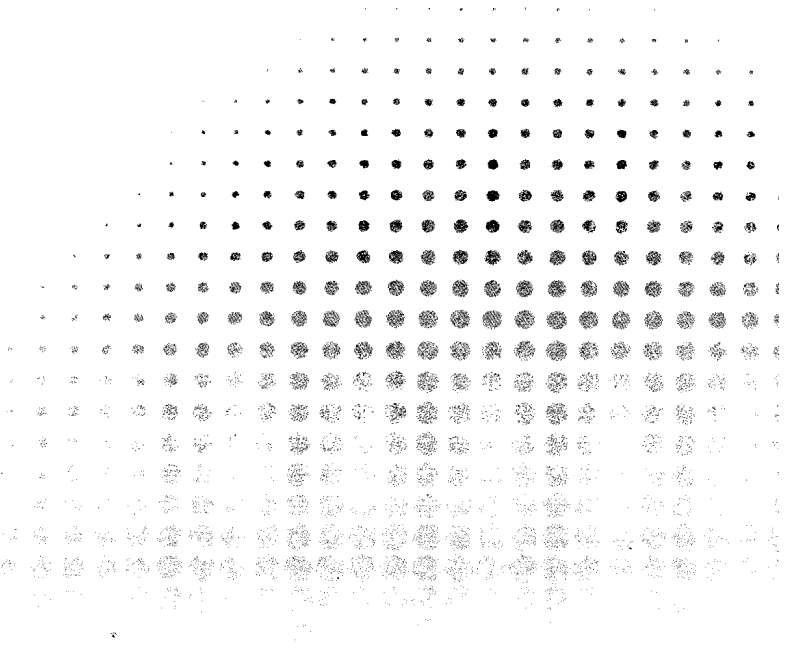
http://en.wikipedia.org/wiki/Comparison_of_command_shells 站点对所有主要的可用 shell 进行了详尽的对比。

本章标志着本书偏理论部分的结束。本书第 II 部分将深入介绍 Unix 与 Linux 中各种可用的工具。第 III 部分将更密切地关注特定的 shell 特性，为典型的实际任务提供各种实用脚本。

第 II 部分

系统工具使用与扩展诀窍

- 第 12 章 文件操作
- 第 13 章 文本操作
- 第 14 章 系统管理工具



第 12 章

文件操作

第 II 部分将更深入地介绍典型 Unix/Linux 系统中可用的通用工具。本章涵盖一些日常使用的特性, 以及一些不是很常用的特性, 并为使用这些工具的方式提供一些提示与技巧。第 II 部分的这几章偶尔会让读者参考手册页中那些晦涩选项的更多信息, 因为在书中将手册页重印出来实在没有必要。很多 GNU 命令, 特别是 `coreutils` 包, 会在 `info` 页中包含比手册页中更好的对于命令的解释。如果系统中安装了 `info`, 下面的语法可以阅读到关于 `stat`(`coreutils` 包的一部分) 的信息。

```
info coreutils 'stat invocation'
```

本章还介绍了输入与输出重定向。它是对于基本重定向所要理解的概念。但在使用 `here` 文档将输入重定向至标准输入以及从循环与其他外部命令转移输入与输出的情况下, 重定向会变得稍微有些复杂。

12.1 stat

`stat` 是一个很不可思议的实用工具。它让 `shell` 用户具有了自 Unix 诞生起 C 程序员便具有的访问能力。系统调用 `stat(2)` 调用操作系统内核中恰当的文件系统驱动程序, 并查询存储在文件的 `inode` 中的细节。`stat(1)` 命令将内核调用显示到用户空间。这给予了 `shell` 程序员很多关于文件的有用信息, 否则必须从 `ls` 这样的程序的输出中提取这些信息。而 `stat` 会给出整个 `inode`, 而且可以直接查询 `inode` 的各个方面。

```
$ stat /etc/nsswitch.conf
File: `/etc/nsswitch.conf'
Size: 513          Blocks: 8          IO Block: 4096          regular file
Device: 805h/2053d Inode: 639428   Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 0/ root)  Gid: ( 0/ root)
Access: 2011-03-30 19:35:48.000000000 +0100
Modify: 2010-06-05 19:52:18.000000000 +0100
Change: 2010-06-05 19:52:18.000000000 +0100
```


这非常有趣，而且从其中还可以分析出一些相关细节。同时 `stat` 接受很多种格式选项，可以将输出精确地限定为所需信息。这意味着我们可以创建自己的超级 `ls` 命令，它能以任何希望的方式显示任何需要的信息。

```
$ stat -c "%a %U %G %n" /etc/nsswitch.conf
644 root root /etc/nsswitch.conf
$ stat -c "%n is owned by %U and has permissions %a (%A)" /etc/nsswitch.conf
/etc/nsswitch.conf is owned by root and has permissions 644 (-rw-r--r--)
$
```

RPM 包格式使用一个规范文件来对包进行描述。该文件的开头是一个简单的“字段：值”设置列表。这些确实非常简单，如下所示是 OpenSSH 规范文件中的一小段：

```
Name          : openssh
Version       : %{version}%{cvs}
Release      : %{release}
Group        : System/Network
```

RPM 从文件列表开始越来越复杂。尽管可以为所有文件指定默认的属主、属组与权限设置，但实际上较为复杂的包都会对不同文件进行不同设置。可以对设置进行手动修改，但也可以更精确(且更容易)地扫描已有的经过恰当配置的系统，然后读取每个文件的属性。`stat` 是按照 RPM 规范文件所需的格式进行精确格式化的理想工具。也就是说，格式应当是 `%attr (permissions,owner,group) filename`。按照最佳实践，尽管 RPM 会安装到 `/opt` 中，但该包实际上是由非特权用户从打包者自己的主目录下构建的。可以很容易地在 `stat` 输出格式中的文件名之前添加斜线(`/%n` 而不是 `%n`)来伪装成根目录。



可从
wrox.com
下载源代码

```
$ pwd
/home/steve/rpm
$ find . -ls
155660 4 drwxr-xr-x 3 steve steve 4096 Mar 23 12:54 .
155661 4 drwxr-xr-x 3 steve steve 4096 Mar 28 15:31 ./opt
155662 4 drwxr-xr-x 4 myapp myapp 4096 Mar 30 12:52 ./opt/myapp
155663 4 drwxr-xr-x 2 myapp myapp 4096 Mar 30 12:52 ./opt/myapp/etc
155664 4 -rw----- 1 myapp myapp 12 Mar 30
12:52 ./opt/myapp/etc/myapp.conf
155665 4drwxr-xr-x 2 myapp myapp 4096 Mar 30 12:52./opt/myapp/bin
155666 8 -rwxr-x--- 1 myapp myapp 4368 Mar 30
12:52 ./opt/myapp/bin/myapp
155669 4-rwxr-xr-x 1 steve steve 111 Mar 30 12:54./packager.sh
$ cat packager.sh
#!/bin/bash

find opt/myapp -print | while read filename
do
    stat -c "%a (%a,%U,%G) /%n" "$filename"
done
$ ./packager.sh | tee -a myapp.rpm
%attr (755,myapp,myapp) /opt/myapp
%attr (755,myapp,myapp) /opt/myapp/etc
```

```
%attr (600,myapp,myapp) /opt/myapp/etc/myapp.conf
%attr (755,myapp,myapp) /opt/myapp/bin
%attr (750,myapp,myapp) /opt/myapp/bin/myapp
$
```

packager.sh

12.2 cat

`cat` 是 Unix/Linux 工具箱中最简单的工具之一。它的使用非常广，并且能完成比我们所想的关于它的基本描述要多得多的功能。我们将从 `cat`(concatenate 的缩写)的基本操作开始。例如，命令 `cat file1 file2` 将显示作为参数传递的所有文件的内容。

```
$ cat file1
this is file1

it has four lines.
the second line is blank.
$ cat file2
this is file2

it has six lines in all (three blank),

of which this is the sixth.
$ cat file1 file2
this is file1

it has four lines.
the second is blank.
this is file2

it has six lines (three blank)

of which this is the sixth.
$
```



这些文件可以不是文本文件。`cat` 也能用于二进制文件。

这看起来不是一个特别有用的功能，而且它确实可能是 `cat` 命令最不常用的用法！本节介绍一些使用 `cat` 实用工具各种不同功能的脚本。在设计 `shell` 脚本时，知道一些像这样的基本功能是有好处的，因为使用系统工具操作文件内容比起自己在 `shell` 脚本中完成这些任务要高效且容易得多。因此，熟悉一些不太常用的功能可以使 `shell` 脚本编程更加轻松。

12.2.1 行号标记

在处理多行代码、配置文件或很多其他文本文件时，显示每行的行数会比较有用。`-n`

选项可以完成这一任务。

```
$ cat -n file1 file2
 1 this is file1
 2
 3 it has four lines.
 4 the second is blank.
 5 this is file2
 6
 7 it has six lines (three blank)
 8
 9
10 of which this is the sixth.
$
```

要对每个文件单独标记行号，则需要对每个文件执行 `cat` 命令。命令可以用分号隔开，如下所示：

```
$ cat -n file1 ; cat -n file2
 1 this is file1
 2
 3 it has four lines.
 4 the second is blank.
 1 this is file2
 2
 3 it has six lines (three blank)
 4
 5
 6 of which this is the sixth.
$
```

12.2.2 处理空白行

文件中的空白行通常没有什么用处。它们在配置文件中是为了增强可读性，但对 `shell` 没有任何作用。下面的调用使用 `-s` 标志保留空白行来保持段落的原来模样，但会缩减重复的空白行以避免浪费空间。

```
$ cat -s file1 file2
this is file1

it has four lines.
the second is blank.
this is file2

it has six lines (three blank)

of which this is the sixth.
$
```

我们可以选择只对非空白行进行行号标记。这与上面代码中的 `cat -n` 非常相似，但它忽略了空白行：

```
$ cat -b file1 file2
  1 this is file1

  2 it has four lines.
  3 the second is blank.
  4 this is file2

  5 it has six lines (three blank)

  6 of which this is the sixth.
$
```

12.2.3 非打印字符

不是所有文件都是文本文件，但也可以使用 `cat` 来处理这些文件。下面的文件包含控制字符。如果使用 `cat` 来查看，则只提示(使用#符号)控制字符。

```
$ cat file3
This is file3. It contains various non-printing
characters, like the tab in this line,
and the #control#codes in this line.
$
```

使用 `-v` 标志可以更清楚地显示文件中的实际 ASCII 字符，它可以显示大多数非打印字符。使用 `-T` 标志会将 `tab` 显式地显示成 `^I`，而不是在行内展开。

```
$ cat -vT file3
This is file3. It contains various non-printing
characters, like the tab ^Iin this line,
and the ^Bcontrol^Dcodes in this line.
$
```



`^B` 与 `^D` 字符分别表示 ASCII 字符 1 与 3。在适当的上下文中，它们分别表示报头开始与文本结束。但在上面这种情况下，它们表示数据损坏。`cat` 命令能在其他没有被损坏的文本周围清楚地显示这一点。

如果一行的末尾有额外的空格，那么知道该行在何处结束也很有用。`-e` 标志会在每行的实际末尾处放置一个 `$` 符号。在下面的代码中，`file2` 的最后一行末尾有一些额外的空格。

```
$ cat -e file2
this is file2$
$
it has six lines (three blank)$
```

```
$  
$  
of which this is the sixth.  $  
$
```

12.3 cat 的反转词 tac

另一个受 cat 启发而来的有用的实用程序是 tac。这又是另一个单词游戏，跟 yacc、bash、GNU 以及很多其他类似的冷幽默如出一辙。

```
$ tac file1 file2  
the second is blank.  
it has four lines.  
  
this is file1  
of which this is the sixth.  
  
it has six lines (three blank)  
  
this is file2  
$
```

尽管翻转文件的内容不是很常用，但可以有一些比较微妙的用途。向文件追加数据很容易，但要数据置于前端，tac 则非常方便。注意，(tac alpha.txt ; echo Bravo ; echo Alpha) 必须在子 shell 中完成，以便让 3 个命令的全部输出都写入到 tempfile。

```
$ cat alpha.txt  
Delta  
Echo  
Foxtrot  
Golf  
Hotel  
$ ( tac alpha.txt ; echo Bravo ; echo Alpha ) > tempfile  
$ cat tempfile  
Hotel  
Golf  
Foxtrot  
Echo  
Delta  
Bravo  
Alpha  
$ tac tempfile > alpha.txt  
$ cat alpha.txt  
Alpha  
Bravo  
Delta
```

```
Echo
Foxtrot
Golf
Hotel
$ rm tempfile
$
```

12.4 重定向

在处理文件输入与输出时，有另外一类输入与输出，即重定向。Unix 的结构隐含了一个水流的比喻，就像通过管道与尖括号的数据流一样。数据从管道左端流向右端，还能被箭头导向。多个箭头表示追加而不是覆盖(或创建)。

12.4.1 重定向输出：单个大于符号(>)

含有单个箭头的 `command > filename` 这样的结构在文件不存在的情况下会创建文件。如果文件确实存在，则会将其长度截短为 0，但文件的 `inode` 信息依然保留。这一结构可用于写入日志文件、创建数据文件，以及执行最通用的文件创建与打开任务。如果无法写入文件，则整个命令行会失败，并且不执行任何操作。下面这个简单的脚本说明，如果文件不存在则创建它，反之则截短它。`date` 命令显示在第二次的时候将不同的数据写入到文件中，但原始内容不见了。



尽管在文件已经存在的情况下权限不会发生变化，但如果文件不存在，则会用 `umask(2)` 的当前值为依据并使用标准权限和所有权来创建文件。



可从
wrox.com
下载源代码

```
$ cat create.sh
#!/bin/bash

LOGFILE=/tmp/log.txt

function showfile
{
    if [ -f "${1}" ]; then
        ls -l "${1}"
        echo "--- the contents are:"
        cat "${1}"
        echo "--- end of file."
    else
        echo "The file does not currently exist."
    fi
}

echo "Testing $LOGFILE for the first time."
showfile $LOGFILE
```

```

echo "Writing to $LOGFILE"
date > $LOGFILE

echo "Testing $LOGFILE for the second time."
showfile $LOGFILE

sleep 10

echo "Writing to $LOGFILE again."
date > $LOGFILE

echo "Testing $LOGFILE for the third and final time."
showfile $LOGFILE

$ ./create.sh
Testing /tmp/log.txt for the first time.
The file does not currently exist.
Writing to /tmp/log.txt
Testing /tmp/log.txt for the second time.
-rw-rw-r-- 1 steve steve 29 Mar 28 14:45 /tmp/log.txt
--- the contents are:
Mon Mar 28 14:45:52 BST 2011
--- end of file.
Writing to /tmp/log.txt again.
Testing /tmp/log.txt for the third and final time.
-rw-rw-r-- 1 steve steve 29 Mar 28 14:46 /tmp/log.txt
--- the contents are:
Mon Mar 28 14:46:02 BST 2011
--- end of file.
$

```

create.sh

12.4.2 追加：双大于符号(>>)

与截短不同，一对大于符号会向已存在的文件追加内容。与单个大于符号结构相同的是，如果文件不存在，则会创建一个文件。如果没有创建或追加文件的权限，则整个命令行会失败且不执行。将上面的脚本进行一些修改会有不同的结果。与 `create.sh` 和 `append.sh` 脚本的唯一区别在于 `echo` 语句显示 `Appending to` 而不是 `Writing to`，`date` 命令使用双大于符号，如下所示：

```
date >> $LOGFILE
```



两个大于符号中间不能有空格。它们必须是 `>>`，而不能是 `> >`。后者会导致语法错误而失败。

此次，在运行之前将 `/tmp/log.txt` 文件删除，`append.sh` 脚本依旧会在第一次运行时创建文件。然而第二次运行时，脚本向文件进行追加，而不是对其截短。



```
$ cat append.sh
#!/bin/bash

LOGFILE=/tmp/log.txt

function showfile
{
    if [ -f "${1}" ]; then
        ls -l "${1}"
        echo "--- the contents are:"
        cat "${1}"
        echo "--- end of file."
    else
        echo "The file does not currently exist."
    fi
}

echo "Testing $LOGFILE for the first time."
showfile $LOGFILE

echo "Appending to $LOGFILE"
date >> $LOGFILE

echo "Testing $LOGFILE for the second time."
showfile $LOGFILE

sleep 10

echo "Appending to $LOGFILE again."
date >> $LOGFILE

echo "Testing $LOGFILE for the third and final time."
showfile $LOGFILE
$ ./append.sh
Testing /tmp/log.txt for the first time.
The file does not currently exist.
Appending to /tmp/log.txt
Testing /tmp/log.txt for the second time.
-rw-rw-r-- 1 steve steve 29 Mar 28 14:53 /tmp/log.txt
--- the contents are:
Mon Mar 28 14:53:04 BST 2011
--- end of file.
Appending to /tmp/log.txt again.
Testing /tmp/log.txt for the third and final time.
-rw-rw-r-- 1 steve steve 58 Mar 28 14:53 /tmp/log.txt
--- the contents are:
Mon Mar 28 14:53:04 BST 2011
Mon Mar 28 14:53:14 BST 2011
--- end of file.
$
```

append.sh

这特别有用，因为它同时还保留了文件中原有的任意格式的其他数据。我们使用任意文本来填充文件，这说明了已有内容绝不会受到影响。它还说明了文件被追加时，其权限不会发生变化。

```
$ echo "Hello, this is some test data." > /tmp/log.txt
$ chmod 600 /tmp/log.txt
$ ls -l /tmp/log.txt
-rw----- 1 steve steve 31 Mar 28 14:57 /tmp/log.txt
$ ./append.sh
Testing /tmp/log.txt for the first time.
-rw----- 1 steve steve 31 Mar 28 14:57 /tmp/log.txt
--- the contents are:
Hello, this is some test data.
--- end of file.
Appending to /tmp/log.txt
Testing /tmp/log.txt for the second time.
-rw----- 1 steve steve 60 Mar 28 14:58 /tmp/log.txt
--- the contents are:
Hello, this is some test data.
Mon Mar 28 14:58:00 BST 2011
--- end of file.
Appending to /tmp/log.txt again.
Testing /tmp/log.txt for the third and final time.
-rw----- 1 steve steve 89 Mar 28 14:58 /tmp/log.txt
--- the contents are:
Hello, this is some test data.
Mon Mar 28 14:58:00 BST 2011
Mon Mar 28 14:58:10 BST 2011
--- end of file.
$
```

12.4.3 输入重定向：单个小于符号(<)

`command < filename` 结构使用与 `>` 相反的重定向，它对输入进行重定向，而不是输出。有些时候，比起使用管道从左端向命令输入数据，使用输入重定向这种方式向命令输入数据会更加方便。它的一个很常见的用法是，将来自文本文件的输入传给 `while` 循环。这使得循环可以连续地读取文件中的行，然后根据情况对它们进行操作。下面这个简单的脚本从标准输入读取主机名，然后尝试执行 `ping` 操作。



可从
wrox.com
下载源代码

```
$ cat readloop.sh
#!/bin/bash

while read -p "Host to check: " hostname
do
    if [ -z "$hostname" ]; then
        echo "Quitting due to blank input"
        break
    fi
done
```

```

fi
ping -c1 -w1 $hostname > /dev/null 2>&1
if [ "$?" -eq "0" ]; then
    echo "Contact made with $hostname"
else
    echo "Failed to make contact with $hostname"
fi
done

$ ./readloop.sh
Host to check: localhost
Contact made with localhost
Host to check: example.com
Contact made with example.com
Host to check: nosuch.example.com
Failed to make contact with nosuch.example.com
Host to check:
Quitting due to blank input
$

```

readloop.sh

在非交互式地运行时，`read` 命令不会显示提示符。同样，循环不会读取空白行，因为首先会遇到 EOF 标志，它会导致 `while read` 测试失败。所以就失去了允许用户输入空白行退出的交互式便利性。该脚本可以通过将标准输入重定向至文件来运行，如下所示：

```

$ cat hosts.txt
declan
192.168.0.1
localhost
$ ./readloop.sh < hosts.txt
Failed to make contact with declan
Contact made with 192.168.0.1
Contact made with localhost
$

```

向输入追加空白行意味着 `[-z "$hostname"]` 测试的成功，并且循环在分析 `www.example.com` 之前退出。这确实在一定程度上改变了脚本的运行方式。不会对 `www.example.com` 这个 URL 进行测试，因为读取了空白行且循环退出。注意，此次显示了消息 `Quitting due to blank input`，而在此前没有显示，因为没有执行过 `[-z "$hostname"]` 这个测试。

```

$ echo >> hosts.txt
$ echo www.example.com >> hosts.txt
$ cat hosts.txt
declan
192.168.0.1
localhost

```

← `hosts.txt` 现在包含一个额外的空白行与 `www.example.com`。它们会被忽略。
`www.example.com`

```
$ ./readloop.sh < hosts.txt
Failed to make contact with declan
Contact made with 192.168.0.1
Contact made with localhost
Quitting due to blank input
$
```

重定向也能在文件中进行。修改最后一行来读取 `done < hosts.txt`，这样对循环进行硬编码使其总是从文件 `hosts.txt` 中读取。循环还可以从 `$1`、`$HOSTS` 以及任何其他所需的内容中读取。修改版的脚本读取定义过的 `$HOSTS` 变量，否则读取 `$1`。



可从
wrox.com
下载源代码

```
$ cat readloop2.sh
```

```
#!/bin/bash
```

```
HOSTS=${HOSTS:-$1}
```

```
while read -p "Host to check: " hostname
do
    if [ -z "$hostname" ]; then
        echo "Quitting due to blank input"
        break
    fi
    ping -c1 -w1 $hostname > /dev/null 2>&1
    if [ "$?" -eq "0" ]; then
        echo "Contact made with $hostname"
    else
        echo "Failed to make contact with $hostname"
    fi
done < $HOSTS
```

```
$ cat hosts2.txt
```

```
example.com
```

```
www.example.com
```

```
$ ./readloop2.sh hosts2.txt
```

```
Contact made with example.com
```

```
Contact made with www.example.com
```

```
$ HOSTS=hosts3.txt
```

```
$ export HOSTS
```

```
$ cat $HOSTS
```

```
bad.example.com
```

```
steve-parker.org
```

```
$ ./readloop2.sh
```

```
Failed to make contact with bad.example.com
```

```
Contact made with steve-parker.org
```

```
$
```

readloop2.sh

12.4.4 here 文档：双小于符号(<< EOF)

<<EOF 语法不是从文件读取，而是从当前标准输入获取内容，直到遇到 EOF 标志。

不需要使用字面文本 EOF。pling、bang 或者感叹号(!) 经常用来表示 EOF。无论在<<与一行的末尾之间使用何种定界文本，定界文本都必须文本行自己来提供。<<EOF 与 EOF 标志本身之间的文本就是 here 文档，因为这个非外部文件以内联的形式提供给 shell 会话(或者是更常见的 shell 脚本)。扩展变量、保留\$(command)与反引号并进行算术扩展，这些便是脚本的余下部分所完成的任务。这个例子展示了标准的 here 文档。它接收命令行输入，然后发送到指定的 Web 服务器。



```
$ cat heredoc.sh
#!/bin/bash
HOST=$1
shift
PORT=80
COMMAND=${@:-HEAD /}

echo "Sending \"${COMMAND}\" to $HOST port $PORT"
netcat ${HOST} ${PORT} <<EOF
${COMMAND}

EOF
echo "Done!"

$ ./heredoc.sh example.com
Sending "HEAD /" to example.com port 80
HTTP/1.0 302 Found
Location: http://www.iana.org/domains/example/
Server: BigIP
Connection: close
Content-Length: 0

Done!
$ ./heredoc.sh www.iana.org HEAD http://www.iana.org/domains/example/ HTTP/1.0
Sending "HEAD http://www.iana.org/domains/example/ HTTP/1.0" to www.iana.org port 8
0
HTTP/1.1 200 OK
Date: Mon, 28 Mar 2011 21:35:20 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Wed, 09 Feb 2011 17:13:15 GMT
Content-Length: 2945
Connection: close
Content-Type: text/html; charset=UTF-8

Done!
$
```

heredoc.sh

here 文档的内容被原模原样地发送。将其置于 shell 脚本中可能会带来不便，因为缩进会变得一团糟。我们可以选择向接收程序发送缩进(无论是否接受)，或者将所有 here 文档的内容(与 EOF 定界符)左对齐。bash 可以避免这一点。添加一个连字符(<<-EOF)后，脚本

会在 here 文档或 EOF 标志之前忽略开头的制表符。下面的脚本对上面的脚本稍微进行了一些重写，说明了如何用来改进演示效果与可读性。在第一次尝试时，因为在一行的开头没有发现 EOF，所以脚本被损坏了。



可从
wrox.com
下载源代码

```
$ cat manyhosts.sh
```

```
#!/bin/bash
```

```
function readhost
```

```
{
```

```
    HOST=$1
```

```
    shift
```

```
    PORT=80
```

```
    COMMAND="HEAD http://${HOST}/ HTTP/1.0"
```

```
    echo "Sending \"${COMMAND}\" to $HOST port $PORT"
```

```
    netcat ${HOST} ${PORT} <<EOF
```

```
        ${COMMAND}
```

```
    EOF
```

```
    echo "Done!"
```

```
}
```

```
for host in $@
```

```
do
```

```
    readhost $host
```

该命令不起作用。一行的开头没有找到 EOF。

```
done
```

```
$ ./manyhosts.sh example.com example.org
```

```
./manyhosts.sh: line 21: warning: here-document at line 11 delimited by end-of-file  
(wanted `EOF')
```

```
./manyhosts.sh: line 22: syntax error: unexpected end of file
```

```
$ sed -i s/"<<EOF"/"<<-EOF"/1 manyhosts.sh
```

```
$ ./manyhosts.sh example.com example.org
```

```
Sending "HEAD http://example.com/ HTTP/1.0" to example.com port 80  
HTTP/1.0 302 Found
```

```
Location: http://www.iana.org/domains/example/
```

```
Server: BigIP
```

```
Connection: close
```

```
Content-Length: 0
```

这里的 sed 只是将<<EOF
替换为<<-EOF。

现在脚本可以用于缩进了。

```
Done!
```

```
Sending "HEAD http://example.org/ HTTP/1.0" to example.org port 80  
HTTP/1.0 302 Found
```

```
Location: http://www.iana.org/domains/example/
```

```
Server: BigIP
```

```
Connection: close
```

```
Content-Length: 0
```

```
Done!
```

```
$
```

manyhosts.sh

12.5 dd

`dd` 命令使一个文件或设备中的比特流向另一个文件或设备。它也向 `stderr` 报告读写记录的次数与整个过程花费的时间。它是用来监视存储设备性能的理想工具。如果怀疑设备或路径偶尔较慢，按一定规律运行下面这样的脚本，可以在一天的任何时刻捕获到一些信息。

```
# dd if=/dev/sda1 of=/dev/null bs=1024k count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 16.8678 s, 63.7 MB/s
#
```

`dd` 的语法有些晦涩。`if`=指定输入文件，`of`=指定输出文件。`bs`=指定复制的块大小，`count`=指定要复制的块的数量。用到的设备会对性能带来极大的影响。极端的例子是 `/dev/zero`(读取最快)、`/dev/null`(写入最快)以及 `/dev/urandom`(这是一个读取非常慢的设备，因为它必须等待系统中有足够的熵来产生随机数)。

存储控制器中的缓存会对速度产生巨大的影响。如果正在做性能测试(特别是在 Linux 下)，Linux 系统缓存也可以非常大。每次读取前执行 `echo 3 > /proc/sys/vm/drop_caches`，以确保缓存被完全清空。



```
$ cat checkpaths.sh
```

```
#!/bin/sh
```

```
DEV1=${1:-sday}
```

```
DEV2=${2:-sdu}
```

```
DEV3=${2:-sdcc}
```

```
DEV4=${2:-sddg}
```

```
EXPECTED=350
```

```
cd /var/tmp
```

```
rm -f dd.pid
```

```
for DEV in $DEV1 $DEV2 $DEV3 $DEV4
```

```
do
```

```
    dd if=/dev/$DEV of=/dev/null bs=8192 count=1000000 2>&1|grep -w copied \
    >> dd.$DEV &
```

```
    echo $! >> dd.pid
```

```
done
```

```
sleep $EXPECTED
```

```
CHILDREN=2
```

```
while [ "$CHILDREN" -gt "0" ]
```

```
do
```

```
    echo "`date`: I have $CHILDREN children"
```

```

    sleep 5
    CHILDREN=`ps hfp $(cat dd.pid) | wc -l`
Done

MAILOUT=0
for SECONDS in `awk '{ print $6 }' dd.$DEV1 dd.$DEV2 dd.$DEV3 dd.$DEV4 | \
    cut -d"." -f1`
do
    if [ "$SECONDS" -gt "$EXPECTED" ]; then
        MAILOUT=1
    fi
done

if [ "$MAILOUT" == "1" ]; then
    for DEV in $DEV1 $DEV2 $DEV3 $DEV4
    do
        msg=`cat dd.$DEV`
        logger -t storagespeed "Path Comparison: $DEV :$msg"
    done

    echo "It should take no more than $EXPECTED seconds to read 8Gb from a device." \
        "It took:\n`grep . dd.$DEV1 dd.$DEV2 dd.$DEV3 dd.$DEV4`" | \
        mailx -s "Slow I/O on `uname -n`" storage@example.com
fi

```

checkpaths.sh

```

# grep storagespeed /var/log/messages
/var/log/messages:Mar 27 07:17:17 node42 storagespeed: Path
Comparison: sday :8192000000 bytes (8.2 GB) copied, 277.018 seconds,
29.6 MB/s
/var/log/messages:Mar 27 07:17:17 node42 storagespeed: Path
Comparison: sdu :8192000000 bytes (8.2 GB) copied, 428.881 seconds, 19.1
MB/s
/var/log/messages:Mar 27 07:17:17 node42 storagespeed: Path
Comparison: sdcc :8192000000 bytes (8.2 GB) copied, 176.26 seconds, 46.5
MB/s
/var/log/messages:Mar 27 07:17:17 node42 storagespeed: Path
Comparison: sddg :8192000000 bytes (8.2 GB) copied, 550.755 seconds,
14.9 MB/s

```

在这个例子中，磁盘设备/dev/sdu 与/dev/sddg 的性能都比较差，分别是 19.1Mbps 与 14.9Mbps。在 logger 工具中使用 storagespeed 标记可以将这一信息记录到系统日志文件中。脚本还自动向适当的邮箱发送电子邮件、报告情况并给出所有相关信息。存储器方面的专家在接收到这封电子邮件后可以对其进行调查。

```

Subject: Slow I/O on node42
From: root <root@node42.example.com>
Date: Sun 27 Mar 2011 07:17:17 +0100

```

To: storage@example.com

```
It should take no more than 350 seconds to read 8Gb from a device. It took:
dd.sday:8192000000 bytes (8.2 GB) copied, 277.018 seconds, 29.6 MB/s
dd.sdu:8192000000 bytes (8.2 GB) copied, 428.881 seconds, 19.1 MB/s
dd.sdcc:8192000000 bytes (8.2 GB) copied, 176.26 seconds, 46.5 MB/s
dd.sddg:8192000000 bytes (8.2 GB) copied, 550.755 seconds, 14.9 MB/s
```

12.6 df

df 报告每个挂载的文件系统的未使用磁盘空间总量。按照惯例，报告采用的单位是KB。但 GNU df 也可以更人性化地显示，如 1024KB 显示为 1MB、1024MB 显示为 1GB，依此类推。较新版本的 sort 可以根据这种类型的输出排序，例如它知道 1GB 是大于 900MB 的。但是在编写本书时，这个功能的使用还不是很广泛，所以最安全的做法是使用 KB 作为报告的单位，使用 sort -k 排序。

下面这个简洁的脚本使用 df 来判断哪个文件系统的可用剩余空间最大，并判断是否足够用来存储某个特定映像。这可以用于计划安装或者管理等工作。如果服务器根本没有用于安装的文件系统，则在安装之前需要进行一些补救工作。然而，即使/var 被占满，如果脚本在/home 中找到足够空间，管理员还是可以选择在/home 目录下进行安装。



可从
wrox.com
下载源代码

```
$ cat freespace.sh
```

```
#!/bin/bash
```

```
required=${1:-2131042}
```

```
preferred=${2:-/var}
```

```
available=`df -k /var | awk '{ print $4 }' | tail -1`
```

```
if [ "$available" -gt "$required" ]; then
```

```
    echo "Good news. There is sufficient space in ${preferred}:"
```

```
    df -h $preferred
```

```
else
```

```
    echo "Bad news. There is not enough space in ${preferred}:"
```

```
    df -h $preferred
```

```
    echo
```

```
    echo "Looking in other filesystems..."
```

```
    fs=`mktemp`
```

```
    df -k -x nfs | sort -k4 -n | awk '{ print $4,$6 }' | grep -v "Available" | \
```

```
        while read available filesystem
```

```
    do
```

```
        if [ "$available" -gt "$required" ]; then
```

```
            echo "Good news: $filesystem has $available Kb" | tee $fs
```

```
        fi
```

```
    done
```

```
    if [ ! -s $fs ]; then
```

```
        echo "No filesystems were found with sufficient free space."
```

```
    exit 1
```



```

fi
rm -f $fs
fi
$ ./freespace.sh 3105613 /var
Bad news. There is not enough space in /var:
Filesystem      Size Used Avail  Use% Mounted on
/dev/sda5        28G  27G  417M   99%  /

Looking in other filesystems...
Good news: /home/steve has 3387316 Kb
$ ./freespace.sh 14502 /var
Good news. There is sufficient space in /var:
Filesystem      Size Used Avail  Use% Mounted on
/dev/sda5        28G  27G  417M   99%  /
$

```

freespace.sh

12.7 mktemp

mktemp 经常用来创建临时文件，并且保证文件名是唯一的。很多脚本使用 `/tmp/programname.$$` 创建临时文件，其中 `$$` 是返回当前运行进程的 PID 的特殊变量。这在通常情况下是满足需要的，但也不尽然。例如，如果我们从 `/tmp/programname.$$` 中读写，而另一个不怀好意的用户在 `/tmp` 中创建很多文件——`programname.1`、`programname.2` 等。这样创建 `/tmp/programname.$$` 的操作就会失败(可能不太容易检查)，于是脚本将数据写入到属于不受信任用户的文件中，或者从不受信任用户提供的文件中读取数据。

真正健壮的解决方案是，必须在返回文件名之前检查文件名没有被使用，然后再创建文件。**mktemp** 就实现了这一功能，按照用户要求创建文件或目录。

mktemp 默认在 `/tmp` 目录中创建一个文件，然后在标准输出中返回文件名。然而，我们可以使用指定的模板来创建文件。如果定义了 `$TMPDIR`，则默认模板是 `$TMPDIR/tmp.XXXXXXXXXX` 或者 `/tmp/tmp.XXXXXXXXXX`——也就是 `/tmp/tmp` 加上 10 个随机生成的字符(大小写字母与数字)，但 `mktemp /var/tmp/helloXXX` 会在 `/var/tmp` 中创建一个以 `hello` 开头且后面有 3 个随机字符的文件。



可从
wrox.com
下载源代码

```

$ ls -l `mktemp`
-rw----- 1 steve steve 0 Oct  8 16:47 /tmp/tmp.9e0M1DJrFW
$
$ cat mktemp.sh
#!/bin/sh

TEMPFILE=`mktemp` || exit 1
ls -l $TEMPFILE
echo "This is definitely my temporary file" > $TEMPFILE
cat $TEMPFILE
rm -f $TEMPFILE
$ ./mktemp.sh

```

```
-rw----- 1 steve steve 0 Oct 8 16:49 /tmp/tmp.FqsOiJihvK
This is definitely my temporary file
$
```

我们还可以指定后缀。指定方法是使用--suffix 标志或者在模板尾部使用 X 以外的其他字符。下面的例子通过参数--suffix .txt 创建了一个.txt 文件。

```
$ cat mktemp.sh
#!/bin/sh

TEMPFILE=`mktemp --suffix .txt` || exit 1
ls -l $TEMPFILE
echo "This is definitely my temporary file" > $TEMPFILE
cat $TEMPFILE
rm -f $TEMPFILE
$ ./mktemp.sh
-rw----- 1 steve steve 0 Oct 8 16:49 /tmp/tmp.AqaySc7mOc.txt
This is definitely my temporary file
$
```



mktemp 创建的文件或目录所在的目录必须存在，否则 mktemp 会失败。
mktemp 不会创建父目录(但 mkdir -p 会创建父目录)。

```
$ cat mktemp.sh
#!/bin/sh

TEMPDIR=`mktemp -d` || exit 1
echo "This is a file in my temporary directory" > $TEMPDIR/file1
echo "This is another file in my temporary directory" > $TEMPDIR/file2
ls -la $TEMPDIR
rm -rf $TEMPDIR
$ ./mktemp.sh
total 16
drwx----- 2 steve steve 4096 Oct 8 16:50 .
drwxrwxrwt 10 root root 4096 Oct 8 16:50 ..
-rw-rw-r-- 1 steve steve 41 Oct 8 16:50 file1
-rw-rw-r-- 1 steve steve 47 Oct 8 16:50 file2
$
```

mktemp.sh



mktemp 创建的文件的最大权限为 0600——也就是说，文件属主可读可写，但没有任何其他权限。如果 umask 比 0600 更加严格，则使用更严格的权限。

12.8 join

join 是根据两个不同文件中的公共键来组合文件的实用程序。两个文件必须按键进行

排序才能使用 `join`。但这通常会被调整，即使意味着从原始输入创建临时文件。另一个限制是，输入与输出文件都必须具有共同的定界符。如果能接受这些限制条件，那么 `join` 会很有用的工具。使用与第 13 章的 `paste` 一节相同的输入数据，`join` 可以用来组合不同的数据库。下面的代码包含两个文件：`hosts` 与 `ethers`。首先对第二个键(主机名)进行排序，以便使它们顺序相同。`join` 命令按照主机名将这两个文件组合起来。

```
$ cat hosts
127.0.0.1      localhost
192.168.1.5    plug
192.168.1.10   declan
192.168.1.11   atomic
192.168.1.13   goldie
192.168.1.227  elvis

$ cat ethers
0a:00:27:00:00:00 plug
01:00:3a:10:21:fe declan
71:1f:04:e3:1b:13 atomic
01:01:8d:07:3a:ea goldie
01:01:31:09:2a:f2 elvis

$ sort -k2 ethers > ethers.sorted
$ sort -k2 hosts > hosts.sorted
$ join -j2 -a1 hosts.sorted ethers.sorted
atomic 192.168.1.11 71:1f:04:e3:1b:13
declan 192.168.1.10 01:00:3a:10:21:fe
elvis 192.168.1.227 01:01:31:09:2a:f2
goldie 192.168.1.13 01:01:8d:07:3a:ea
localhost 127.0.0.1
plug 192.168.1.5 0a:00:27:00:00:00
$
```

此处的公共键是两个文件中的第二个字段，通过 `-j2` 指定。`-a1` 标志告诉 `join` 显示第一个文件中的全部内容，即使在第二个文件中没有匹配项的情况下。这可以用来找出哪些主机在 `ethers` 中没有对应项。本例中只有 `localhost` 没有 `ethers` 对应项。

12.9 install

`install` 是增强型的 `cp`。它可以设置文件权限、创建目录、进行备份，以及执行更多其他操作，而且都是从单个命令行来完成。顾名思义，它编写出来是用于安装软件的脚本的，但也能用于其他目的。

因为文件的已有副本可能已经被安装，所以 `install` 在安装新文件之前也能够进行备份。这是通过 `-b` 标志来指定的，它会在新的文件名末尾使用波浪号(~)来创建备份文件。

`--backup` 选项可以指定其他的备份策略，如下所示：

- `simple` 或者 `never`：只是默认的波浪号(~)备份。
- `numbered` 或者 `t`：编号备份，如 `~1~`、`~2~` 等。

- existing 或者空白: simple 或者 numbered, 哪个使用过就用哪个(默认是 simple)。
- none 或者 off: 不进行备份。

-m 标志对模式进行指定。格式为八进制(普通文件使用 0750, rwx、r-x 与 --- 分别用于属主、属组与其他用户)或者符号格式(如 -m 'u=rwx,g=r')。注意, 该标志对父目录不适用。父目录的创建权限是 755。

-o 与 -g 标志告诉 install 文件属于哪个属主与属组。这些标志只能被 root 用户使用, 因为非特权用户没有执行这些操作的权限。-v 标志要求 install 详细地告知它所执行的操作。这作为一种反馈可以用来观察进行了哪些操作。

下面这个脚本展示了一个简单应用程序的 Makefile 与安装程序。安装程序本身被封装在 Makefile 的内部程序逻辑中, 所以 make install 命令将实现智能化的安装, 而不用将 mkdir、cp、chown 与 chmod 命令放到 Makefile 中。



```
steve@goldie:~$ cd myapp
steve@goldie:~/myapp$ ls
installer.sh Makefile myapp.c myapp.conf
steve@goldie:~/myapp$ cat Makefile
```

```
all:    myapp.c
        $(CC) -o myapp myapp.c
```

```
clean:  myapp
        rm -f myapp
```

```
install:    myapp ←———— install 目标是运行下面的 installer.sh.
            ./installer.sh /opt/myapp myapp myapp
```

```
steve@goldie:~/myapp$ cat installer.sh
```

```
#!/bin/bash
```

```
ROOTDIR=${1:-/opt/myapp}
```

```
OWNER=${2:-myapp}
```

```
GROUP=${3:-myapp}
```

```
# Create bin and opt directories. Parents will be 755; if run
```

```
# as root, their ownership will be root:root. (any suid/sgid will be preserved)
```

```
install -v -m 755 -o $OWNER -g $GROUP -d $ROOTDIR/bin $ROOTDIR/etc
```

```
if [ "$?" -ne "0" ]; then
```

```
    echo "Install: Failed to create directories."
```

```
    exit 1
```

```
fi
```

```
# install the binary itself
```

```
install -b -v -m 750 -o $OWNER -g $GROUP -s myapp $ROOTDIR/bin
```

```
if [ "$?" -ne "0" ]; then
```

```
    echo "Install: Failed to install the binary"
```

```
    exit 2
```

```
fi
```

```
# Install the configuration file, only read-writeable by the owner.
```

```
install -b -v -m 600 -o $OWNER -g $GROUP myapp.conf $ROOTDIR/etc
```

```
if [ "$?" -ne "0" ]; then
```

```

    echo "Install: Failed to install the config file"
    exit 3
fi

echo "Install: Succeeded."
steve@goldie:~/myapp$ make ← 软件可以用非特权用户身份来构建。
cc -o myapp myapp.c
steve@goldie:~/myapp$ su - ← 然后由 root 用户安装。
Password: root_password
root@goldie:~# cd ~steve/myapp
root@goldie:/home/steve/myapp# make install
./installer.sh /opt/myapp myapp myapp
`myapp' -> `/opt/myapp/bin/myapp' (backup: `/opt/myapp/bin/myapp~')
`myapp.conf' -> `/opt/myapp/etc/myapp.conf' (backup: `/opt/myapp/
etc/myapp.conf~')

Install: Succeeded.
root@goldie:/home/steve/myapp# ls -lR /opt/myapp ← 在相关目录中使用 ls
/opt/myapp/:                                列出原始文件(日期为
total 8                                     一月份)与代替它们的
drwxr-xr-x 2 myapp myapp 4096 Mar 28 11:52 bin 新文件(日期为三月份)。
drwxr-xr-x 2 myapp myapp 4096 Mar 28 11:52 etc 注意,目录的时间戳也
                                              被更新了。

/opt/myapp/bin:
total 16
-rwxr-x--- 1 myapp myapp 4368 Mar 28 11:52 myapp
-rwxr-x--- 1 myapp myapp 4368 Jan 12 09:21 myapp~

/opt/myapp/etc:
total 8
-rw----- 1 myapp myapp 12 Mar 28 11:52 myapp.conf
-rw----- 1 myapp myapp 12 Jan 16 17:34 myapp.conf~
root@goldie:/home/steve/myapp# exit
$

```

简洁的安装看起来有些差异。本例显示安装文件创建了中间文件。同样,关于备份的注释也没有了。下面的代码显示了安装执行的这一部分。

```

root@goldie:/home/steve/myapp# make install
./installer.sh /opt/myapp myapp myapp
install: creating directory `/opt/myapp'
install: creating directory `/opt/myapp/bin'
install: creating directory `/opt/myapp/etc'
`myapp' -> `/opt/myapp/bin/myapp'
`myapp.conf' -> `/opt/myapp/etc/myapp.conf'
Install: Succeeded.
root@goldie:/home/steve/myapp# exit

```

myapp、Makefile 和 installer

12.10 grep

grep 是 Unix/Linux 工具箱中极其重要的一部分。它在输入中搜索匹配传递给它的正则

表达式的文本行。其最简单(与最常用)的用法是用来搜索某个固定字符串。在这种情况下, **grep** 的标志比它的正则表达式功能更加重要。本章后面的 12.10.2 节将介绍 **grep** 的正则表达式语法。**grep** 是非常常用的命令, 以至于我们能在本书的很多代码示例中找到它, 甚至是在它没有被直接使用的时候。本节给出了一些比较常用的用法, 并介绍 **grep** 的功能。

12.10.1 **grep** 标志

grep 的 4 个最常用标志是 **-i**(不区分大小写的搜索)、**-l**(只列出匹配文件的名称)、**-w**(只进行全字匹配)与 **-v**(反选, 仅列出与模式不匹配的行)。另一个不太常用但更有用的标志是 **-e**。它能用来在单个命令上传递多个搜索模式。**grep -e** 作为首选方式取代 **egrep** 来同时搜索多个模式。下面的例子使用了一个有着固定结构的通讯录(**contacts.txt**)。其中每个联系人有 4 行与之相关的内容, 并且每一行包含一个字段。除此之外, 其他都不是固定的——文件中可以有任意其他文本, 而且空格也可以按任意方式填充。我们可以非常高效地对这 4 个标志进行测试。

```
$ head -4 contacts.txt
Name:Steve Parker
Phone:44 789 777 2100
Email:steve@steve-parker.org
Web:http://steve-parker.org/
$ grep -i sTEve contacts.txt ←—— -i 将搜索指定为不区分大小写。
Name:Steve Parker
Email:steve@steve-parker.org
Web:http://steve-parker.org/
$ grep -l Steve * ←—— 这一行说明当前目录中还有另一个包含 Steve 单词的文件。
contacts.txt
users.txt
$ grep -v -e 0 -e Steve -e http contacts.txt ← 多个<-e 表达式>都被-v 反选。
不匹配这些表达式的行被列出来。

Name:Richard Stallman
Email:rms@stallman.org

Name:Linus Torvalds
Email:torvalds@osdl.org
```

GNU **grep** 的另外 3 个(不太为人所知的)可用标志是 **-A**、**-B** 和 **-C**。这些标志分别给出匹配行之前、之后与周围(上下文)的一定数量的文本行。它们可以用来高效地给出搜索结果的上下文。

```
$ grep -A2 Steve contacts.txt ←—— 匹配行的后两行是电话号码与电子邮件地址。
Name:Steve Parker
Phone:44 789 777 2100
Email:steve@steve-parker.org
$ grep -B4 steve-parker contacts.txt ←匹配行之前的 4 行构成了该联系人的完整记录。
Name:Steve Parker
Phone:44 789 777 2100
```

```
Email:steve@steve-parker.org
Web:http://steve-parker.org/
$
```

12.10.2 grep 正则表达式

当传递了正则表达式时, **grep** 使用下面的规则进行匹配。它将匹配最长的可能模式(贪婪匹配)。

?	前项可选且最多匹配一次。
*	前项匹配零到多次。
+	前项匹配一到多次。
{n}	前项匹配 n 次。
{n,}	前项匹配 n 到多次。
{,m}	前项最多匹配 m 次。
{n,m}	前项最少匹配 n 次, 最多匹配 m 次。

下面的 **contacts.sh** 脚本使用正则表达式 `^Name:.*${name}` 搜索以标签 **Name:** 开头的任意行, 后面的 `*` 表示匹配任意数目的任意字符, 它最后会包含用户输入的字符串。为了方便, **grep** 也使用 `-i` 来进行不区分大小写的搜索。



可从
wrox.com
下载源代码

```
$ cat contacts.sh
#!/bin/bash
CONTACTS=contacts.txt
PS3="Search For: "

select task in "Show All Information" "Show Phone" "Show Email" "Show Web"
do
    name=""
    if [ "$REPLY" -le "4" ] && [ "$REPLY" -ge "1" ]; then
        while [ -z "$name" ]
        do
            read -p "Enter a name to search for: " name
        done
        case $REPLY in
            1)
                grep -A3 -i "^Name:.*${name}" $CONTACTS
                ;;
            2)
                grep -A1 -i "^Name:.*${name}" $CONTACTS | cut -d: -f2-
                ;;
            3)
                grep -A2 -i "^Name:.*${name}" $CONTACTS | \
                grep -e "^Email:" -e "^Name:" | cut -d: -f2-
                ;;
            4)
                grep -A3 -i "^Name:.*${name}" $CONTACTS | \
                grep -e "^Web:" -e "^Name:" | cut -d: -f2-
```

```

        ;;
    esac
fi
done

$ ./contacts.sh
1) Show All Information   3) Show Email
2) Show Phone            4) Show Web
Search For: 1
Enter a name to search for: steve
Name:Steve Parker
Phone:44 789 777 2100
Email:steve@steve-parker.org
Web:http://steve-parker.org/
Search For: 4
Enter a name to search for: s
Steve Parker
http://steve-parker.org/
Richard Stallman
http://stallman.org/
Linus Torvalds
http://www.cs.helsinki.fi/~torvalds
Search For: ^C
$

```

contacts.sh

第一次搜索找到 `contacts.txt` 文件中唯一一个 Steve。第二次搜索在文件中任意位置查找 `^Name:.*s`。对该正则表达式执行 `grep -A3 -i`，它会匹配 Steve、Stallman 与 Linus Torvalds。每个匹配行与其随后的 3 行文本被输出，但在显示之前，`grep -e` 只挑选出姓名与电子邮件这两行。最后，出于演示目的，在显示每个人的单个项时，将字段名从输出中截去。

12.11 split

在管理与转移文件的时候，通常都会有能够转移的文件大小的限制。无论是 2MB 或 10MB 限制的电子邮件系统，还是 4GB 限制的 FAT 文件系统，或者将大量数据保存到 4.7GB 的 DVD 中，总会有一些问题。当习惯了使用灵活、强大，且绝对有能力处理比以上还要大得多的文件的 OS 来解决这些问题时，我们会发现很多主流的架构都没有现代 Linux 或 Unix 系统强大。

解决这一问题的方法是使用 `split` 实用程序。尽管它可以基于行号对文件进行分割，但它最常用将大文件分割成较小的普通数据块。`-b` 标志告诉 `split` 以这种方式运行。我们可以用各种单位指定数据块的大小，如 K、M、G、T、P、E、Z，甚至 Y。它们之间是 1 024 的倍数关系。添加后缀 B(KB、MB 与 GB 等)后将变成 1 000 的倍数关系。

使用默认选项被广泛地认可。这时 `split` 创建一系列名为 `xaa`、`xab`、`xac` 等(直到 `xax`、`xay`、`xaz`、`xba`、`xbb`、`xbc`、`xbd`……)的文件。然而，`split` 确实有一些更友好的选项。`-d` 标

志使用数字而不是字母——x01、x02 与 x03 而不是 xaa、xab 与 xac——并且可以自定义前缀而不用字母 x。在前缀末尾加上下划线是一个比较好的习惯，这可以使更容易辨认出后缀。



后缀的默认大小为两个字符。如果文件要分割成 100 多个数据块，则需要使用 -a 选项指定更长的后缀长度：-a3 表示多达 1 000 个块，-a4 表示多达 10 000 个块等。不过，split 不会为我们计算后缀长度，而是在它用完所有可用后缀之后给出 output file suffixed exhausted 消息。

看看下面这个例子。它将一个 Solaris 10 Update 9 的虚拟机映像(刚好大于 4GB)分割成 5 个最大 1GB(1024*1024*1024 字节)的文件，使用 Sol10u9 前缀和数字后缀(01~05)。时间戳显示这些操作花费大概 4 分钟：

```
$ ls -l
-rwxrwxr-x 1 steve steve 4332782080 Oct 20 11:16 Solaris 10 u9.vdi
$ ls -lh
-rwxrwxr-x 1 steve steve 4.1G Oct 20 11:16 Solaris 10 u9.vdi
$ split -b 1G -d Solaris\ 10\ u9.vdi Sol10u9_
$ ls -l
total 8470776
-rw-rw-r-- 1 steve steve 1073741824 Dec  1 10:20 Sol10u9_00
-rw-rw-r-- 1 steve steve 1073741824 Dec  1 10:21 Sol10u9_01
-rw-rw-r-- 1 steve steve 1073741824 Dec  1 10:23 Sol10u9_02
-rw-rw-r-- 1 steve steve 1073741824 Dec  1 10:24 Sol10u9_03
-rw-rw-r-- 1 steve steve  37814784 Dec  1 10:24 Sol10u9_04
-rwxrwxr-x 1 steve steve 4332782080 Oct 20 11:16 Solaris 10 u9.vdi
$ ls -lh
total 8.1G
-rw-rw-r-- 1 steve steve 1.0G Dec  1 10:20 Sol10u9_00
-rw-rw-r-- 1 steve steve 1.0G Dec  1 10:21 Sol10u9_01
-rw-rw-r-- 1 steve steve 1.0G Dec  1 10:23 Sol10u9_02
-rw-rw-r-- 1 steve steve 1.0G Dec  1 10:24 Sol10u9_03
-rw-rw-r-- 1 steve steve  37M Dec  1 10:24 Sol10u9_04
-rwxrwxr-x 1 steve steve 4.1G Oct 20 11:16 Solaris 10 u9.vdi
$
```

我们可以使用 cat 命令将这些文件组合在一起，且 cat 用于二进制文件与用于文本文件是一样的。下面的例子证明确实如此。diff 命令的结果说明生成的文件与原文件是一样的。

```
$ cat Sol10u9_* > Sol10.vdi
$ ls -lh
total 12.2G
-rw-rw-r-- 1 steve steve 1.0G Dec  1 10:20 Sol10u9_00
-rw-rw-r-- 1 steve steve 1.0G Dec  1 10:21 Sol10u9_01
-rw-rw-r-- 1 steve steve 1.0G Dec  1 10:23 Sol10u9_02
-rw-rw-r-- 1 steve steve 1.0G Dec  1 10:24 Sol10u9_03
```

```

-rw-rw-r-- 1 steve steve 37M Dec 1 10:24 Sol10u9_04
-rwxrwxr-x 1 steve steve 4.1G Oct 20 11:16 Solaris 10 u9.vdi
-rwxrwxr-x 1 steve steve 4.1G Dec 20 10:49 Sol10.vdi
$ diff Solaris\ 10\ u9.vdi Sol10.vdi
$ echo $?
0
$

```

12.12 tee

还有一些我们未曾介绍的工具在脚本中可以简单明了地完成一些有用的工作，**tee** 就是其中之一。它将输入传递给 **stdout**，但同时也会写入到文件中。使用 **-a** 标志，它会对文件追加。我们可以用两个 **echo** 语句向标准输出与日志文件进行写操作。在这种情况下，需要确保对一个输出行的任意修改都会在匹配行中重复。个人经验告诉我们不会发生这样的事情。这使得在输出与日志文件有细微差别的时候更加令人不解。下面的脚本足够简单，但两行之间的 3 个区别不太明显。而更糟糕的是，一旦发现了它们的差异，要确定哪一行有错误也不是显而易见的。



可从
wrox.com
下载源代码

```

$ cat bad.sh
#!/bin/bash

for i in `seq -w 1 10` 14 19 13
do
    j=`expr $i \* $i`
    echo "`date`: I am `basename $0` and I can count to ${i}, " \
        "which is not particularly impressive, especially as I get a bit" \
        "confused after ten. I do know that the prime factors of ${i} squared are" \
        "`factor $j | cut -d. -f2 | cut -c2- | tr ' ' 'x' | sed s/"`"$$/"/(none)"/1`" \
        "which is a bit more impressive."
    echo "`date`: I am `basename $0` and I can count to ${i}, " \
        "which is not particularly impressive, especially as I get a bit" \
        "confused after ten. I do know that the prime factors of ${i} squared are" \
        "`factor $i | cut -d: -f2 | cut -c1- | tr ' ' 'x' | sed s/"`"$$/"/(none)"/1`" \
        "which is a bit more impressive." >> /tmp/count.log
    sleep 10
done
$

```

bad.sh



这里引入的 3 个错误是 **factor \$i**、**cut -d.** 与 **cut -c1-**。显示该脚本的输出没有任何意义，因为 **stdout** 与 **count.log** 都不正确，而且格式有误。

两次运行同一个命令行带来的第二个问题是，如果命令(或管道)运行时间很长，则在这样的一个循环中，总共消耗的时间会加倍。更好的办法是使用同一个命令行，将其输出

同时转向标准输出与日志文件。

下面这个简单的脚本计算一组 CD 的 ISO 映像的 MD5 校验和，并将校验和写入一个日志文件中。由于使用了 `tee`，脚本可以在创建直接作为 `md5sum --check` 输入的日志文件的同时也向用户产生有用输出，而且完全不用重复代码。



```
$ cat tee.sh
#!/bin/bash
LOGFILE=/tmp/iso.md5

> $LOGFILE
find /iso -type f -name "*.iso*" -print | while read filename
do
    echo "Checking md5sum of $filename"
    md5sum "$filename" | tee -a $LOGFILE
done

$ ./tee.sh
Checking md5sum of /iso/SLES-11-DVD-i586-GM-DVD1.iso.gz
5d7a7d8a3e296295f6a43a0987f88ecd /iso/SLES-11-DVD-i586-GM-DVD1.iso.gz
Checking md5sum of /iso/Fedora-14-i686-Live-Desktop.iso.gz
6d07f574ef2e46c0e7df8b87434a78b7 /iso/Fedora-14-i686-Live-Desktop.iso.gz
Checking md5sum of /iso/oi-dev-147-x86.iso.gz
217cb2c9bd64eecb1ce2077fffb2238 /iso/oi-dev-147-x86.iso.gz
Checking md5sum of /iso/solaris/sol-11-exp-201011-text-x86.iso.gz
05505ece2efc4a046c0b51da71b37444 /iso/solaris/sol-11-exp-201011-text-x86.iso.gz
Checking md5sum of /iso/solaris/sol-10-u9-ga-x86-dvd.iso.gz
80f94ce0f8ab3093aelbeafb8aef75d8 /iso/solaris/sol-10-u9-ga-x86-dvd.iso.gz
$ cat /tmp/iso.md5
5d7a7d8a3e296295f6a43a0987f88ecd /iso/SLES-11-DVD-i586-GM-DVD1.iso.gz
6d07f574ef2e46c0e7df8b87434a78b7 /iso/Fedora-14-i686-Live-Desktop.iso.gz
217cb2c9bd64eecb1ce2077fffb2238 /iso/oi-dev-147-x86.iso.gz
05505ece2efc4a046c0b51da71b37444 /iso/solaris/sol-11-exp-201011-text-x86.iso.gz
80f94ce0f8ab3093aelbeafb8aef75d8 /iso/solaris/sol-10-u9-ga-x86-dvd.iso.gz
$
```

tee.sh

12.13 touch

`touch` 是创建文件或者更新文件时间戳的程序，而且不用向文件进行任何实际的写入操作。初看起来似乎没什么用处，但实际上却非常有用。一些工具会简单地检查某个文件是否存在。当 Solaris 重启并找到名为 `/reconfigure` 的文件后，它会自动执行一次重新配置的重启。很多初始化脚本会在 `/var/lock/subsys` 中创建文件，表示正在运行某个服务。这个目录中需要的通常也就是一个空文件。它还可以用来将文件更新为当前时间戳。`make` 命令基于源文件与二进制文件的更新程度来修改运行方式。`find` 命令可以通过设定只报告比某个给定的文件样本更新的文件。

`touch` 也可以用来使行为表现得与实际不太一样。这对于使用原始时间戳存储文件的备份实用程序非常有用。它还能用于其他目的,包括一些不值得称道的目的。如果系统管理员偶然对文件进行了错误的修改,对于他来说通过不修改时间戳来掩盖痕迹是有可能的。

```
# ls -l /etc/hosts
-rw-r--r-- 1 root root 511 Feb 25 12:14 /etc/hosts
# grep atomic /etc/hosts
192.168.1.15      atomic
# stat -c "%y" /etc/hosts
2011-02-25 12:14:12.000000000 +0000
# timestamp=`stat -c "%y" /etc/hosts`
# echo $timestamp
2011-02-25 12:14:12.000000000 +0000
# sed -i s/192.168.1.15/192.168.1.11/1 /etc/hosts
# ls -l /etc/hosts
-rw-r--r-- 1 root root 511 Mar 31 16:32 /etc/hosts
# date
Thu Mar 31 16:32:29 BST 2011
# grep atomic /etc/hosts
192.168.1.11      atomic
# touch -d "$timestamp" /etc/hosts
# ls -l /etc/hosts
-rw-r--r-- 1 root root 511 Feb 25 12:14 /etc/hosts
#
```

然而,不是所有这些都与看上去一样。`touch` 无法用来修改文件的更改时间,也就是 `inode` 细节更新的时间。`touch` 命令的目的是修改 `inode` 细节,所以这个狡猾的管理员终究无法掩盖修改痕迹。

```
# stat -c "%z" /etc/hosts
2011-03-31 16:32:29.000000000 +0100
#
```

12.14 find

`find` 工具对文件系统进行全面搜索来查看文件(包括它们的 `inode`),并可以对它们进行各种测试,然后对匹配指定条件的文件进行某些操作。尽管 `find` 的语法理解起来可能比较困难,但使用 `find` 对整个文件系统进行全面搜索几乎总是比自己编写等价的功能在效率上要高得多。



如果只需要用文件名查找某个特定文件,并且 `updatedb` 命令已在系统中运行(通常使用 `cron` 定期调用),那么命令 `locate name-of-file` 几乎会立即返回一个结果。这取决于当前文件系统中最新的文件名数据库。但如果该数据库存在,而且我们只对文件名感兴趣,那么 `locate` 可能比 `find` 要快几乎不知道多少倍。

find 的参数基本上被分解为表达式与动作两部分。最常用的表达式如表 12-1 所示。

表 12-1 find 表达式

表 达 式	用 途
-maxdepth levels	对文件系统树的搜索只深入到第 levels 层
-mount (或者-xdev)	不跨越不同的文件系统
-anewer filename、-cnewer filename 或者-newer filename	查找比参考文件 filename 的访问(-anewer)时间、改变(-cnewer)时间或修改(-newer)时间更迟的文件
-mmin n 或者-mtime n	查找 <i>n</i> 分钟(-mmin)或 <i>n</i> 天(-mtime)前修改的文件
-uid u 或者-user u	查找属于用户 ID(-uid)或用户名(-user)u 的文件
-gid g 或者-group g	查找属于组 ID(-gid)或组名(-group)g 的文件
-nouser 或者-nogroup	查找不匹配属主/属组名称的文件
-name n 或者-iname n	查找名称匹配 <i>n</i> (-iname 表示不区分大小写)的文件
-perm -g=w	查找设置了组可写位的文件(不管其他权限位)
-perm o=r	查找权限为 0500 的文件(只有属主可读)
-size n	读取大小为 <i>n</i> 的文件(后缀为 b、k、M 和 G，整个列表与其他后缀也可以使用)
-size +n 或者-size -n	查找大小大于(+n)或小于(-n)n 的文件
-type t	查找类型为 <i>t</i> 的文件，其中 <i>t</i> 可以是 d(目录)、l(链接)、f(文件)或者 b、c、p、s 与 D

这些表达式可以组合起来。尽管此处的代码示例应当演示如何组合表达式，但还是请参考 find(1)手册页中的一些例子。

find 常用的有用动作要容易理解一些。-print 简单地将匹配文件输出来。-ls 功能相同，但输出方式与 ls 命令非常相似。-print0 将在第 14 章的 14.16 节介绍。还有-exec 动作，将在本节稍候介绍。

增量索引器可以使用 find 来识别自上次运行以来被修改过的文件。例如，一个基于静态文件的网站可能在一天中有各个贡献者进行更新，但如果每次上传新文件时都要重新索引数据库，那么效率会比较低。每到晚上再重新索引所有文件也较为低效，因为一些文件可能很大且从来都不会变化。通过对 find 使用-newer 选项，索引器会定期运行(可能每天一次)，识别新的或更新过的文件，然后在晚上对它们进行重新索引。下面这个脚本识别适合索引的文件。它使用过滤器 find . -type f -newer \$LASTRUN 实现这一功能。脚本只查找修改时间比\$LASTRUN 文件的时间戳更晚的常规文件(不包括目录、块设备驱动程序等)。在第一次调用或者当\$LASTRUN 文件不存在时，脚本会索引所有文件。在随后的运行中，只有新近更新的文件才会被索引。



可从
wrox.com
下载源代码

```
# cat reindexer.sh
#!/bin/bash
LASTRUN=/var/run/web.lastrun
WEBROOT=/var/www
# be verbose if asked.
START_TIME=`date`

function reindex
{
    # Do whatever magic is required to add this new/updated
    # file to the database.
    add_to_database "$@"
}

if [ ! -f "$LASTRUN" ]; then
    echo "Error: $LASTRUN not found. Will reindex everything."
    # index from the epoch...
    touch -d "1 Jan 1970" $LASTRUN
    if [ "$?" -ne "0" ]; then
        echo "Error: Cannot update $LASTRUN"
        exit 1
    fi
fi

cd $WEBROOT
find . -type f -newer $LASTRUN -print | while read filename
do
    reindex "$filename"
done
echo "Run complete at `date`."
echo "Subsequent runs will pick up only files updated since this reindexing"
echo "which was started at $START_TIME"
touch -d "$START_TIME" $LASTRUN
if [ "$?" -ne "0" ]; then
    echo "Error: Cannot update $LASTRUN"
    exit 1
fi
ls -ld $LASTRUN

# ./reindexer.sh
Error: /var/run/web.lastrun not found. Will reindex everything.
Mon Mar 28 09:19:51 BST 2011: Added to database: ./images/sgp_title_bg.jpg
Mon Mar 28 09:19:52 BST 2011: Added to database: ./images/vcss-blue.gif
Mon Mar 28 09:19:53 BST 2011: Added to database: ./images/shade2.gif
Mon Mar 28 09:19:54 BST 2011: Added to database: ./images/datacentrewide.jpg
Mon Mar 28 09:20:00 BST 2011: Added to database: ./images/shade1.gif
Mon Mar 28 09:20:03 BST 2011: Added to database: ./images/sgp_services.jpg
Mon Mar 28 09:20:04 BST 2011: Added to database: ./images/sgp_content_bg.jpg
[ some output omitted for brevity ]
Mon Mar 28 09:20:51 BST 2011: Added to database: ./services.txt
```

```

Mon Mar 28 09:20:53 BST 2011: Added to database: ./sgp_style.css
Mon Mar 28 09:20:54 BST 2011: Added to database: ./i
Mon Mar 28 09:20:55 BST 2011: Added to database: ./common-services.txt
Run complete at Tue Mar 28 09:20:57 BST 2011.
Subsequent runs will pick up only files updated since this reindexing
which was started at Tue Mar 28 09:19:49 BST 2011
-rw-r--r-- 1 root root 0 Mar 28 09:19 /var/run/web.lastrun
#

```

reindexer.sh

当该脚本在第二天早上运行时，它只会挑选更新过的文件。注意，脚本从 09:04 运行到 09:05，但 `web.lastrun` 文件更新用的时间戳是运行开始的时间 09:04，而不是结束时间 09:05。这使得文件可以在脚本运行时进行更新，而脚本在随后的运行中会把更新过的文件挑选出来。

```

# ./reindexer.sh
Tue Mar 29 09:04:46 BST 2011: Added to database: ./images/shade2.gif
Tue Mar 29 09:04:51 BST 2011: Added to database: ./images/shade1.gif
Tue Mar 29 09:04:58 BST 2011: Added to database: ./services.txt
Tue Mar 29 09:05:01 BST 2011: Added to database: ./sgp_style.css
Run complete at Tue Mar 29 09:05:02 BST 2011.
Subsequent runs will pick up only files updated since this reindexing
which was started at Tue Mar 29 09:04:46 BST 2011
-rw-r--r-- 1 root root 0 Mar 29 09:04 /var/run/web.lastrun
#

```

12.15 find -exec

`find` 的 `-exec` 标志使 `find` 对每个匹配的文件运行给定命令。执行给定命令时，`find` 会将文件名放到占位符 `{}` 所在位置。该命令必须以分号结束。在 `shell` 中使用分号必须经过转义——`\;` 或者 `"`。在下面的脚本中，对找到的每个文件执行 `md5sum`，然后存储在一个临时文件中。该脚本使用 `find -exec` 命令完成这一任务：

```
find "${DIR}" $SIZE -type f -exec md5sum {} \; | sort > $MD5
```



`$SIZE` 变量可选地为 `find` 的标志添加 `-size +0`，因为对一堆长度为 0 的文件执行 `md5sum` 没有太多意义。空文件的 MD5 校验和总是 `d41d8cd98f00b204-e9800998ecf8427e`。

关于 `uniq` 如何过滤结果的详细解释见第 13 章的 13.12 节。简而言之，`-w32` 告诉 `uniq` 只查看校验和；`-d` 告诉 `uniq` 忽略唯一的行，因为它们不可能代表重复的文件。

高效定位重复文件的问题不像刚开始听起来那么简单。对于潜在的 GB 或 TB 级别的数据，使用 `diff` 与所有其他文件进行比较是较为低效的。首先计算每个文件的校验和，代

价相对较高,但随后恰当地使用 `sort` 与 `uniq`,可以相当容易且快速地得到可能的匹配集合。下面脚本中的 `-c` 选项做了一些额外的工作,依然对任何一对具有相同 MD5 校验和的文件执行 `diff` 命令。这不见的是最低效的做法,因为 `diff` 可以快速识别不相同的文件,但在它有把握宣布两个文件确实一样之前必须比较这两个相同文件的每一个字节。鉴于两个不同文件校验和相同的可能性极低,使用 `-c` 选项是否值得取决于使用情况。

如脚本中注释提到的一样, `md5sum` 与很多命令一样可以接收一组文件名作为输入,但在处理的文件数目比内核准备分配给命令参数的个数还多时,则它完全不会运行。因此, `find -exec` 是这一问题的理想解决方案。简单地接收相对较小的文本,在转储到 `$MD5` 之前通过管道送给 `sort`,这样该任务已经接近完成。`find -exec | sort` 这一管道命令周围所有的填充处理实际上只是为了合理性检查与提供更美观的输出。



可从
wrox.com
下载源代码

```
$ cat check_duplicate_files.sh
#!/bin/bash
MD5=`mktemp /tmp/md5.XXXXXXXXXX`
SAMEFILES=`mktemp /tmp/samefiles.XXXXXXXXXX`
matches=0
comparisons=0
combinations=0

VERBOSE=1
SIZE=""
DIR=`pwd`
diff=0

function logmsg()
{
    if [ "$VERBOSE" -ge "$1" ]; then
        shift
        echo "$@"
    fi
}

function cleanup()
{
    echo "Caught signal - cleaning up."
    rm -f ${MD5} ${SAMEFILES} > /dev/null
    exit 0
}

function usage()
{
    echo "Usage: `basename $0` [-e] [-v verbosity] [-c] [-d directory]"
    echo "  -e ignores empty files"
    echo "  -v sets verbosity from 0 (silent) to 9 (diagnostics)"
    echo "  -c actually checks the files"
    exit 2
}
```



```
# Parse options first
while getopts 'ev:l:cd:' opt
do
    case $opt in
        e) SIZE="-size +0 " ;;
        v) VERBOSE=$OPTARG ;;
        d) DIR=$OPTARG ;;
        c) diff=1 ;;
    esac
done

trap cleanup 1 2 3 6 9 11 15

logmsg 3 "`date`: `basename $0` starting."
kickoff=`date +%s`

# Make sure that the temporary files can be created
touch $MD5 || exit 1

start_md5=`date +%s`
logmsg 3 "`date`: Gathering MD5 SUMs. Please wait."
find "${DIR}" $SIZE -type f -exec md5sum {} \; | sort > $MD5
#md5sum `find ${DIR} ${SIZE} -type f -print` > $MD5
# cutting out find is a lot faster, but limited to a few thousand files
done_md5=`date +%s`
logmsg 3 "`date`: MD5 SUMs gathered. Comparing results..."
logmsg 2 "md5sum took `expr $done_md5 - $start_md5` seconds"

uniq -d -w32 $MD5 | while read md5 file1
do
    logmsg 1 "Checking $file1"
    grep "^${md5}" $MD5 | grep -v "^${md5} ${file1}$" | cut -c35- > $SAMEFILES
    cat $SAMEFILES | while read file2
    do
        duplicate=0
        if [ "$diff" -eq "1" ]; then
            diff "$file1" "$file2" > /dev/null
            if [ "$?" -eq "0" ]; then
                duplicate=1
            else
                duplicate=2
            fi
        else
            duplicate=1
        fi
        case $duplicate in
            0) ;;
            1)
                if [ "$VERBOSE" -gt "5" ]; then
                    echo "$file2 is duplicate of $file1"
                fi
            ;;
        esac
    done
done
```

```

else
    echo $file2
fi
;;
2) echo "$file1 and $file2 have the same md5sum" ;;
esac
done
done
endtime=`date +%s`
logmsg 2 "Total Elapsed Time `expr $endtime - $kickoff` seconds."
logmsg 2 "`date`: Done. `basename $0` found $matches matches in
                                $comparisons compar
isons."
logmsg 2 "Compared `wc -l $MD5 | awk '{ print $1 }'` files; that makes
                                for $combina
tions combinations."
rm -f ${MD5} > /dev/null
$

```

check_duplicate_files.sh

`find` 命令产生像下面代码一样的输出。使用管道将这些输出送给 `sort`，显示了\$MD5 文件中最后的结果。

```

$ find . -type f -exec md5sum {} \;
288be591a425992c4247ea5bccd2c929 ./My Photos/DCIM0003.doc
698ef8996726c92d9fbb484eb4e49d73 ./My Photos/DCIM0001.jpg
c33578695c1de14c8deeba5164ed0adb ./My Photos/DCIM0002.jpg
ecca34048d511d1dc01afe71e245d8b1 ./My Documents/doc1.doc
288be591a425992c4247ea5bccd2c929 ./My Documents/cv.odt
619a126ef0a79ca4c0f3e3d061b4e675 ./etc/hosts
d41d8cd98f00b204e9800998ecf8427e ./etc/config.txt
d265cc0520a9a43e5f07ccca453c94f5 ./bin/ls
619a126ef0a79ca4c0f3e3d061b4e675 ./bin/hosts.bak
da5e6e7db4ccbdb62076fe529082e5cd ./listfiles
$ find . -type f -exec md5sum {} \; | sort
288be591a425992c4247ea5bccd2c929 ./My Documents/cv.odt
288be591a425992c4247ea5bccd2c929 ./My Photos/DCIM0003.doc
619a126ef0a79ca4c0f3e3d061b4e675 ./bin/hosts.bak
619a126ef0a79ca4c0f3e3d061b4e675 ./etc/hosts
698ef8996726c92d9fbb484eb4e49d73 ./My Photos/DCIM0001.jpg
c33578695c1de14c8deeba5164ed0adb ./My Photos/DCIM0002.jpg
d265cc0520a9a43e5f07ccca453c94f5 ./bin/ls
d41d8cd98f00b204e9800998ecf8427e ./etc/config.txt
da5e6e7db4ccbdb62076fe529082e5cd ./listfiles
ecca34048d511d1dc01afe71e245d8b1 ./My Documents/doc1.doc
$ find . -type f -exec md5sum {} \; | sort | uniq -d -w32
288be591a425992c4247ea5bccd2c929 ./My Documents/cv.odt
619a126ef0a79ca4c0f3e3d061b4e675 ./bin/hosts.bak

```

快速地扫一眼排序后的输出可以看到，其中 `hosts.bak` 与 `hosts` 的 MD5 校验和相同，`cv.odt` 与 `DCIM0003.doc` 也是。`uniq -d` 命令除去任何真正的唯一项，而只剩下 `cv.odt` 与 `hosts.bak`。这些文件的校验和随后可以在 `$MD5` 文件中找到，可以显示出重复的文件。

```
$ find . -type f -exec md5sum {} \; \  
    | grep 288be591a425992c4247ea5bccd2c929  
288be591a425992c4247ea5bccd2c929 ./My Photos/DCIM0003.doc  
288be591a425992c4247ea5bccd2c929 ./My Documents/cv.odt  
$ find . -type f -exec md5sum {} \; \  
    | grep 619a126ef0a79ca4c0f3e3d061b4e675  
619a126ef0a79ca4c0f3e3d061b4e675 ./etc/hosts  
619a126ef0a79ca4c0f3e3d061b4e675 ./bin/hosts.bak  
$
```

如上文提到的，最后需要提供关于是否对识别出来的匹配文件运行 `diff` 程序的安全级别。

12.16 本章小结

本章主要讨论如何操作文件本身，以及文件与所在文件系统之间的关系。第 13 章将讨论如何对文件中(或其他位置)的文本进行操作。

第 13 章

文 本 操 作

在 Unix 与 Linux 系统中，一切皆文件。其中很大一部分都是纯 ASCII 文本，而且在 Unix 中有很多文本操作的工具。这是工具应当“做一件事并把它做好”这一原则的另一个体现。本章介绍了几种最有名的文本操作工具，还有一些不是那么有名的工具。

这些文本转换过滤器一般能从命令行的 `stdin` 或者指定文件中获取输入，然后将输出发送到 `stdout`。这意味着，尽管它们可以直接用于文件，但通常在管道中更有用。所以，`cut -d: -f7 /etc/passwd | sort | uniq` 会在一个管道中列出所有配置在 `/etc/passwd` 中的不同 shell。

本章将详细介绍一些 GNU/Linux 环境下可用的最好、最常见且最有用的文本操作工具。对于 GNU，大多数这些工具现在都包含在 `coreutils` 包中。曾经的 `fileutils`、`shellutils` 与 `textutils` 都已合并到 `coreutils` 包中。

13.1 cut

`cut` 命令被广泛用于 shell 脚本。它与 `paste` 相反，但这里所说的“剪切”和“粘贴”与 GUI 中将数据转移到剪贴板然后再粘贴回去没有任何关系。`cut` 相对于像 `awk` 这样更加重量级的选择的优势在于它小得多，而且简单得多，因此运行要更快。这似乎无关紧要，但在循环中会有明显区别。`cut` 接收一个或两个参数。在其最简单的形式中，`cut -c n` 只剪切掉输入或者作为参数传递的文件的每行的第 `n` 列，`cut -c m-n` 剪切掉第 `m` 列到第 `n` 列。还可以根据定界符进行剪切，例如，`cut -d: -f5 /etc/passwd` 将从 `/etc/passwd` 的每一行提取由冒号隔开的第 5 个字段。下面的脚本剪切出冒号隔开的第 1 个字段中的文件名，然后是页面中从第一个 `>` 到下一个 `<` 之间的标题。尽管对文件格式理解越好，这种方法就越可靠，但该方法并不是那么完美。



```
$ cat gettitle.sh
#!/bin/bash
```

可从
wrox.com
下载源代码

```
grep "<title>" *.html | while read html
```

```

do
    filename=`echo $html | cut -d: -f1`
    title=`echo $html | cut -d">" -f2- | cut -d"<" -f1`
    echo "$filename = $title"
done
$ grep "<title>" *.html
Aliases.html:<title>Aliases - Bash Reference Manual</title>
ANSI_002dC-Quoting.html:<title>ANSI-C Quoting - Bash Reference Manual</title>
Arrays.html:<title>Arrays - Bash Reference Manual</title>
Bash-Builtins.html:<title>Bash Builtins - Bash Reference Manual</title>
Bash-Features.html:<title>Bash Features - Bash Reference Manual</title>
Bash-POSIX-Mode.html:<title>Bash POSIX Mode - Bash Reference Manual</title>
$ ./getttitle.sh
Aliases.html = Aliases - Bash Reference Manual
ANSI_002dC-Quoting.html = ANSI-C Quoting - Bash Reference Manual
Arrays.html = Arrays - Bash Reference Manual
Bash-Builtins.html = Bash Builtins - Bash Reference Manual
Bash-Features.html = Bash Features - Bash Reference Manual
Bash-POSIX-Mode.html = Bash POSIX Mode - Bash Reference Manual
$

```

getttitle.sh

13.2 echo

大多数人都对 `echo` 命令比较熟悉。除了只是向终端显示文本序列以外，它还有一些其他功能。下面的 `dial1` 与 `dial2` 脚本演示了当脚本执行一些复杂的操作但不知道要花费多长时间时如何向交互式用户提示程序正处于更新状态，且还在执行过程中。

13.2.1 dial1 脚本

“转盘”的第一个实现实际上只显示当前时间，每秒一次，但不会在屏幕上写满时间戳序列。它是通过在给定时间之前向终端发送 `Ctrl-M(^M)` 字符来实现的，并且取消 `echo` 默认的在显示的每行末尾添加 `\n` 的行为。这一默认行为使得随后的输出处于屏幕新的一行开头。

`Ctrl-M` 是回车符(也常称为 `CR` 或 `\r`)，通常有 `\n`(换行符)跟随其后。只是发送 `Ctrl-M` 本身意味着光标回到一行的开头，然后显示日期，最后光标保持原位，而不是走到下一行的开头将处于终端底部的当前行上移。



默认情况下，`bash` 内置命令 `echo` 将反斜线当成常规字符。`echo` 的 `-e` 开关使它将反斜线解释成标记特殊字符序列。该脚本对此没有严格要求，但 `echo` 通常会使用 `-e` 选项。`Bourne echo` 默认会解释反斜线；`echo -e` 可以有效地使 `bash` 的 `echo` 与 `Bourne` 的 `echo` 相同。

`echo` 的默认行为是在显示输入后添加`\n`。这可以使用`-n` 开关关闭。试验这些开关可以很容易看到它们对结果的影响。下面的代码段开了个头。

```
$ echo "this line is followed by a newline"
this line is followed by a newline
$ echo -n "but this one is not."
but this one is not.$
$
$ echo "another way to omit the newline\c"
another way to omit the newline\c
$ echo -e "but it requires the -e switch\c"
but it requires the -e switch$
$
$ echo "backslash is nothing special.\r\n\r\n"
backslash is nothing special.\r\n\r\n
$ echo -e "unless you use the -e switch.\r\n\r\n"
unless you use the -e switch.

$
```

向脚本输入 `Ctrl-M` 字符要使用元字符 `Ctrl-V`。在 `vim` 中，按下 `Ctrl` 键，再按下 `v` 键，会显示一个`^`符号。再按下 `Ctrl` 与 `m` 键，会有一个大写的 `M` 显示在`^`符号后面：`echo -e "^M`date`"`。我们在此对 `cat` 使用了`-v` 开关，使得它将非打印字符显示成可辨认的格式。



可从
wrox.com
下载源代码

```
$ cat -v diall.sh
#!/bin/bash

function stopdial
{
    if [ ! -z "$DIALPID" ]; then
        kill -9 $DIALPID
        unset DIALPID
        echo
    fi
}

function dial
{
    echo -en " "
    while :
    do
        echo -en "^M`date`"
        sleep 1
    done
    echo
}

# on any signal stop the dial subprocess
```

```

trap stopdial `seq 1 63`

echo Starting
echo "`date`: Doing something long and complicated..."

dial &
DIALPID=$!
sleep 10
stopdial

echo "`date`: Finished the complicated bit. That was hard!"
echo Done
$ ./dial1.sh
Starting
Fri Mar 4 17:26:48 GMT 2011: Doing something long and complicated...
Fri Mar 4 17:26:57 GMT 2011
Fri Mar 4 17:26:58 GMT 2011: Finished the complicated bit. That was hard!
Done
$

```

dial1.sh

书面上不是太好说明运行的效果，但在上例的 17:26:48 与 17:26:57 之间，时间在中间一行更新，每秒一次。如果脚本在中途运行中关闭，`trap` 能确保转盘也能关闭。否则，终端会这样持续不断地更新当前时间。这里有一个脚本没有开启 `trap` 的尝试。



我们增加 `sleep` 来稍微腾出一些时间输入显示关于正在进行何种操作的信息。历史命令中也有 `ps` 命令，这样就不必输入太快——如果输入太慢，输入的命令行会被日期弄乱：“`ps Fri Mar 4 17:57:42 -eaf`”。

```

$ ./dial1.sh
Starting
Fri Mar 4 17:57:39 GMT 2011: Doing something long and complicated...
Fri Mar 4 17:57:43 GMT 2011^C
$
$ echo stopped it
stopped it
Fri Mar 4 17:57:51 GMT 2011
Fri Mar 4 17:57:59 GMT 2011
Fri Mar 4 17:58:07 GMT 2011
$ echo it came back
it came back
Fri Mar 4 17:58:15 GMT 2011
$ echo help
help
Fri Mar 4 17:58:19 GMT 2011
$ !ps
ps -eaf|grep dial1.sh

```

```

steve 4214    1 0 17:57 pts/3 00:00:00 /bin/bash ./diall.sh
steve 4239 3699 0 17:58 pts/3 00:00:00 grep diall.sh
Fri Mar 4 17:58:23 GMT 2011
$ kill -9 4214
$

```



dial 函数中最后一个 echo 语句不是必需的, 因为 while 循环永远也不会结束。然而, 将终端回置到一个一致的状态这样的做法可以使终端显示比较美观, 而且如果循环被某个以任意原因终止循环的函数取代, 那么我们会希望在退出函数的时候, 终端看起来能比较美观。在 dial2 脚本中, 该 echo 被移至 stopdial 函数中, 原因是该函数要考虑更多清理屏幕显示的问题。

13.2.2 dial2 脚本

不断更新 date 命令是一种向用户告知某种操作仍然在进行之中的方法。更聪明的做法是使用旋转的 ASCII 字符转盘。用一个字符按顺序显示 dial 数组中的每个字符。这样看起来就好像转盘在不停地顺时针旋转。

该脚本中对 dial 函数的另一处稍进行了修改。函数使用了计数器 \$d 来跟踪要显示的字符。它还发送退格符 Ctrl-H, 而不是回车符 Ctrl-M。与使用 Ctrl-M 不同, 它所带来的好处是, 转盘可以在一行中的任意位置开始, 即可以有一个引导型的 echo(还要使用 -n 开关)引入转盘。



可从
wrox.com
下载源代码

```

$ cat -v dial2.sh
#!/bin/bash

function stopdial
{
    if [ ! -z "$DIALPID" ]; then
        kill -9 $DIALPID
        unset DIALPID
        echo -en "^H"
    fi
}

function dial
{
    dial=('/' '-' '\' '|' '/' '-' '\' '|')
    echo -en " "
    d=0
    while :
    do
        echo -en "^H${dial[$d]}"
        d=`expr $d + 1`
        d=`expr $d % 8` # size of dial[] array
        sleep 1
    done
}

```



```

done
echo
}

# on any signal stop the dial subprocess
trap stopdial `seq 1 63`

echo Starting
echo "`date`: Doing something long and complicated..."

echo -en "Here is a dial to keep you amused: "
dial &
DIALPID=$!
sleep 10
stopdial

echo "`date`: Finished the complicated bit. That was hard!"
echo Done
$ ./dial2.sh
Starting
Fri Mar 4 18:05:51 GMT 2011: Doing something long and complicated...
Here is a dial to keep you amused:
Fri Mar 4 18:06:01 GMT 2011: Finished the complicated bit. That was hard!
Done
$

```

dial2.sh

同样，在书面上很难演示出效果来，但实际上在信息“keep you amused:”之后有一个每秒转动八分之一圈的转盘。`stopdial` 函数中额外的 `echo -en "^H"` 保证光标在转盘上回退一格，然后在原来位置写入一个空格覆盖掉转盘旋转后上次显示的字符。`dial` 函数中最后的 `echo` 也会做同样的清理工作。

13.3 fmt

作为 GNU coreutils 包一部分的 `fmt` 是一个典型的有用但不为人所知的系统工具。其目的是格式化文本行，与第 7 章的 `trimline.sh` 脚本非常相似。除了将文本行分割与合并较短的行外，`fmt` 的 `-p` 选项还有一个非常有用的对于语言来说也是最重要的功能，即使用表示整行注释的标记(如 shell 中的 `#` 符号，或 C/C++ 中的 `//` 符号)。`fmt` 也可以进行一些基本的格式化。`-u` 标志让命令确保每个单词之间有一个空格，句号后面有两个空格。

下面第一个一行脚本对 *Gulliver's Travels* 中的一段摘录运行 `fmt` 命令。如 `cat` 命令所示，原始 `gulliver.txt` 没有真的格式化，但 `fmt -ut` 修正了词语换行、单词间隔甚至缩进。

```

$ cat gulliver.txt
CHAPTER I.

```

```

The author gives some account of himself and family. His first inducements to tr

```

avel. He is shipwrecked, and swims for his life. Gets safe on shore in the country of Lilliput; is made a prisoner, and carried up the country.

My father had a small estate in Nottinghamshire: I was the third of five sons. He sent me to Emanuel College in Cambridge at fourteen years old, where I resided three years, and applied myself close to my studies; but the charge of maintaining me, although I had a very scanty allowance, being too great for a narrow fortune, I was bound apprentice to Mr. James Bates, an eminent surgeon in London, with whom I continued four years.

```
$ fmt -ut gulliver.txt
```

```
CHAPTER I.
```

The author gives some account of himself and family. His first inducements to travel. He is shipwrecked, and swims for his life. Gets safe on shore in the country of Lilliput; is made a prisoner, and carried up the country.

My father had a small estate in Nottinghamshire: I was the third of five sons. He sent me to Emanuel College in Cambridge at fourteen years old, where I resided three years, and applied myself close to my studies; but the charge of maintaining me, although I had a very scanty allowance, being too great for a narrow fortune, I was bound apprentice to Mr. James Bates, an eminent surgeon in London, with whom I continued four years.

```
$
```

第二个一行脚本使用 `-p` 标志调用 `fmt` 对 `shell` 脚本中的注释进行格式化。如果使用 `-s` 标志, `fmt` 可以只将过长的文本行进行分割。如果不使用 `-s`, `fmt` 会将较短的行放在一起。一旦 `fmt` 将一切调整好, 它会确保每行注释都以符号 `#` 开头。非注释行完全保留原样。

```
$ cat code.sh
```

```
#!/bin/bash
```

```
# this is a useful script that does far
```

```
# more than its comments
```

```
# might suggest at first glance.
```

```
#
```

```
# for one thing, it demonstrates the valid use of comments in a shell script in a useful and practical manner.
```

```
# for another, it is very short and concise, unlike these comments, which are really rather rambling at best.
```

```
# so it is useful
```

```
# to have a script
```

```
# like this in your arsenal.
```

```
echo "$@"
```

```
# That was fun.
```

```
# The end.
```

```
# fin
```

```
$ fmt code.sh -p "#"
#!/bin/bash
# this is a useful script that does far more than its comments might
# suggest at first glance.
#
# for one thing, it demonstrates the valid use of comments in a shell
# script in a useful and practical manner. for another, it is very short
# and concise, unlike these comments, which are really rather rambling
# at best. so it is useful to have a script like this in your arsenal.

echo "$@"
# That was fun. The end. fin
$ fmt -p"#" -w 40 code.sh
#!/bin/bash
# this is a useful script that does far
# more than its comments might suggest
# at first glance.
#
# for one thing, it demonstrates the
# valid use of comments in a shell
# script in a useful and practical
# manner. for another, it is very
# short and concise, unlike these
# comments, which are really rather
# rambling at best. so it is useful
# to have a script like this in your
# arsenal.

echo "$@"
# That was fun. The end. fin
$ fmt -p"#" -w 40 -s code.sh
#!/bin/bash
# this is a useful script that does far
# more than its comments
# might suggest at first glance.
#
# for one thing, it demonstrates the
# valid use of comments in a shell
# script in a useful and practical
# manner.
# for another, it is very short and
# concise, unlike these comments, which
# are really rather rambling at best.
# so it is useful
# to have a script
# like this in your arsenal.

echo "$@"
# That was fun.
# The end.
# fin
$
```

比 `fmt` 更强大的工具是 `indent`，它可以使用各种风格对整个 C 程序进行格式化。这些风格包括 Kernighan 与 Ritchie 在他们的开创性著作中使用的以作者名字命名的风格，还有 GNU 风格与 Linux 内核源码的标准风格。这使得开发人员很容易使用各自的首选格式来组合代码，然后以标准格式在多个开发人员之间进行共享。这也有助于将一个项目中的代码重用于另一个项目，而不破坏项目关于编码风格的规定。

13.4 head 和 tail

`head` 和 `tail` 命令也遵循“做一件事并把它做好”的原则。它们用于文本文件或者管道。它们基于逐行的模式，从文件或管道的开头或结尾提取文本行。通过组合这两个工具，我们可以从文件中间提取单独的文本行或者一组文本行。

13.4.1 奖牌脚本

下面第一个脚本将 `head` 与 `tail` 用于 `shuf` 来首先对名字列表进行随机排序，然后从生成的临时文件中提取前 3 个名字。执行一次这样的随机性操作，然后再反复回到保存的结果中。每次挑选出不同的部分，确保结果的一致性，这样两次出现同一个人是不可能的。选出的前 3 个人分别获得金牌、银牌与铜牌。第 4、5、6 名获得进步奖。随机结果的最后一名获得安慰奖。



```
$ cat prizes.sh
#!/bin/bash
PEOPLE=people.txt
temp=`mktemp`

shuf $PEOPLE > $temp
prizes=( Gold Silver Bronze )

position=0
head -3 $temp | while read name
do
    echo "The ${prizes[$position]} prize goes to $name"
    position=`expr $position + 1`
done
echo

echo "There are three runners-up prizes. In alphabetical order, the winners are:"
head -6 $temp | tail -3 | sort
echo
echo "The booby prize goes to `tail -1 $temp`. Bad luck `tail -1 $temp`!"
echo
echo "Congratulations to everybody who participated."
rm -f $temp

$ ./prizes.sh
```

```
The Gold prize goes to Emily
The Silver prize goes to Bethany
The Bronze prize goes to Christopher
```

```
There are three runners-up prizes. In alphabetical order, the winners are:
Anthony
Mary
Daniel
```

```
The booby prize goes to John. Bad luck John!
```

```
Congratulations to everybody who participated.
$
```

prizes.sh

13.4.2 世界杯脚本

第二个使用 `head` 和 `tail` 的脚本稍微复杂一些。然而，它遵循一个类似的原则：对于世界杯足球锦标赛，每个国家被随机分配到一个小组，然后小组里的每个国家都与同组的所有其他国家进行一场比赛。与上一节的奖牌脚本类似，国家通过 `shuf` 进行随机排序，然后 `for group in `seq 1 $NUMGROUPS`` 循环从经过随机排序的文件中提取适当的文本行。如果一组里面有 8 支队伍(如 2010 世界杯的测试数据一样)，那么在第一次迭代中，`grouphead` 是 8，所以 `head` 提取前 8 行。然后，`tail` 获取这 8 行中的最后 8 行。这在第一次运行时没有太多作用，但对于随后的迭代是有意义的。在第二次迭代中，`grouphead` 是 16，所以 `head` 提取前 16 行，然后 `tail` 提取最后 8 行，所以 `/tmp/group2` 得到随机排序文件中的第 9~16 行。第三次迭代时，`grouphead` 是 24，所以 `tail -8` 得到第 17~24 行。第四次迭代时，`grouphead` 是 32，所以 `head` 命令返回文件的全部内容，然后对其运行 `tail -8` 得到第 25~32 行。

一旦该脚本对小组进行了排序，就使用 `pr` 将它们显示为列式形式(节省空间)，且 `arrangegames` 函数安排每个小组的成员都与组内其他成员进行一场比赛。最终，小组赛的顺序被打乱，这样感觉更自然，也让每个队伍在比赛之间有更多的机会休息。最后还是为了节省一些空间，使用 `pr` 排成两列。

```
$ cat worldcup.sh
#!/bin/bash

function arrangegames
{
    played=`mktemp`
    grep -v "^**** Group " /tmp/group${group} | while read team
    do
        # can't play against yourself
        grep -v "${team}$" /tmp/group${group} | \
        grep -v "^**** Group " | while read opponent
        do
            grep "^${opponent} vs ${team}$" $played > /dev/null
            if [ "$?" -ne "0" ]; then
```

```

        echo "$team vs $opponent" | tee -a $played
    fi
done
done
rm -f $played
}

##### Script Starts here #####
TEAMS=teams.txt
RANDOMIZED=`mktemp`
NUMTEAMS=`wc -l $TEAMS | awk '{ print $1 }'`
NUMGROUPS=4

# Each group must have an even number of teams
TEAMSINGROUP=`echo "$NUMTEAMS / $NUMGROUPS" | bc`
echo "scale=1; $TEAMSINGROUP / 2" | bc | grep "\.0$" > /dev/null 2>&1
if [ "$?" -ne "0" ]; then
    echo "$NUMTEAMS does not divide into $NUMGROUPS groups neatly."
    exit 1
fi

shuf $TEAMS > $RANDOMIZED

for group in `seq 1 $NUMGROUPS`
do
    echo "*** Group $group ***" > /tmp/group${group}
    grouphead=`expr $group \* $TEAMSINGROUP`
    head -$grouphead $RANDOMIZED | tail -$TEAMSINGROUP >> /tmp/group${group}
done
echo "Groupings:"
pr -t -m /tmp/group*
echo

for group in `seq 1 $NUMGROUPS`
do
    echo "*** Qualifying games in Group $group ***"
    # Randomizing the order gives the teams more of a break.
    arrangegames $group | shuf | pr -t -c2
    echo
done
$ ./worldcup.sh
Groupings:
*** Group 1 *** *** Group 2 *** *** Group 3 *** *** Group 4 ***
Ghana           Italy           Uruguay         Cameroon
Cote d'Ivoire   Mexico          Denmark         Japan
Greece           United States   Germany         South Korea
Brazil           Chile           Slovakia        Nigeria
Portugal         Netherlands     Spain           Switzerland
New Zealand     Slovenia        Argentina       Honduras
Australia        France          South Africa    Serbia

```

Paraguay	England	Algeria	North Korea
*** Qualifying games in Group 1 ***			
Portugal vs Australia		Brazil vs New Zealand	
Portugal vs New Zealand		Greece vs Paraguay	
Ghana vs Brazil		Cote d'Ivoire vs Portugal	
Cote d'Ivoire vs New Zealand		Greece vs Portugal	
Greece vs Australia		Cote d'Ivoire vs Greece	
Cote d'Ivoire vs Paraguay		Ghana vs Paraguay	
Greece vs Brazil		Brazil vs Portugal	
Brazil vs Paraguay		Cote d'Ivoire vs Australia	
Brazil vs Australia		Cote d'Ivoire vs Brazil	
New Zealand vs Australia		Ghana vs Greece	
New Zealand vs Paraguay		Ghana vs Cote d'Ivoire	
Ghana vs New Zealand		Ghana vs Portugal	
Greece vs New Zealand		Ghana vs Australia	
Australia vs Paraguay		Portugal vs Paraguay	
*** Qualifying games in Group 2 ***			
Italy vs United States		Mexico vs Slovenia	
United States vs England		Italy vs France	
Italy vs England		Slovenia vs France	
Italy vs Mexico		United States vs Slovenia	
France vs England		Netherlands vs Slovenia	
Netherlands vs England		Italy vs Netherlands	
United States vs Chile		Chile vs Netherlands	
Mexico vs United States		Italy vs Chile	
Mexico vs France		Chile vs England	
United States vs Netherlands		Italy vs Slovenia	
Mexico vs Chile		Mexico vs England	
Slovenia vs England		Chile vs France	
Chile vs Slovenia		Mexico vs Netherlands	
United States vs France		Netherlands vs France	
*** Qualifying games in Group 3 ***			
Germany vs South Africa		Argentina vs South Africa	
Argentina vs Algeria		Uruguay vs Spain	
Uruguay vs Denmark		Denmark vs Spain	
Slovakia vs South Africa		Germany vs Algeria	
Germany vs Argentina		Uruguay vs Germany	
Spain vs Argentina		Denmark vs Algeria	
Uruguay vs Argentina		Slovakia vs Algeria	
Spain vs Algeria		Uruguay vs Algeria	
Germany vs Slovakia		Slovakia vs Spain	
South Africa vs Algeria		Uruguay vs Slovakia	
Denmark vs Argentina		Denmark vs Slovakia	
Germany vs Spain		Spain vs South Africa	
Denmark vs Germany		Uruguay vs South Africa	
Denmark vs South Africa		Slovakia vs Argentina	

```

*** Qualifying games in Group 4 ***
Switzerland vs Serbia          Nigeria vs North Korea
South Korea vs North Korea     South Korea vs Nigeria
South Korea vs Honduras        South Korea vs Switzerland
Japan vs Serbia                Nigeria vs Serbia
Nigeria vs Switzerland        Honduras vs North Korea
Cameroon vs North Korea        Switzerland vs North Korea
Cameroon vs South Korea        Cameroon vs Serbia
Serbia vs North Korea          Japan vs Honduras
Japan vs Switzerland           South Korea vs Serbia
Honduras vs Serbia             Nigeria vs Honduras
Cameroon vs Japan              Japan vs Nigeria
Switzerland vs Honduras        Cameroon vs Switzerland
Japan vs South Korea            Cameroon vs Nigeria
Japan vs North Korea            Cameroon vs Honduras
$

```

tail 命令的 **-f** 与 **-F** 选项也非常有用。它们会监视追加到文件中的文本行。**-F** 与 **-f** 的不同之处是，使用 **-F** 时文件不一定要存在，或者运行 **tail** 过程中如果文件被删除或截短，**tail** 会等待文件再出现。将其用于交互式监控日志文件特别有用，但也能用在 **shell** 脚本中。尤其是将 **tail -F** 进程转到后台，这样文件一旦有更新就会显示给用户，但脚本可以继续执行。下面这个脚本在运行时监控 **Apache** 访问情况与错误日志文件，所以用户可以在 **Web** 服务器自身记录日志的同时查看日志结果。

-n0 开关使 **tail** 不报告文件的任何已有内容(默认情况是显示最后 10 行)。**-f** 标志使 **tail** 对文件进行密切监视，显示新到的每一行。

```

# cat apache.sh
#!/bin/bash

tail -n0 -f /var/log/apache2/access.log &
access=$!
tail -n0 -f /var/log/apache2/error.log &
error=$!

echo "Requesting HEAD of /..."
printf "HEAD / HTTP/1.0\n\n" | netcat localhost 80
echo
echo
echo "---- `date`"
sleep 10
echo "---- `date`"
echo "Requesting /nofile..."
printf "GET /nofile HTTP/1.0\n\n" | netcat localhost 80

kill -9 $access
kill -9 $error
# ./apache.sh
Requesting HEAD of /...

```



```
[Wed Mar 02 13:01:45 2011] [error] [client 127.0.0.1] PHP Notice: Undefined variable: panelFile in /home/steve/sgp/newweb/php/layoutStart.php on line 25
[Wed Mar 02 13:01:45 2011] [error] [client 127.0.0.1] PHP Notice: Undefined variable: panelTitle in /home/steve/sgp/newweb/php/layoutStart.php on line 25
HTTP/1.1 200 OK
Date: Wed, 02 Mar 2011 13:01:45 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: PHP/5.3.3-7
Vary: Accept-Encoding
Connection: close
Content-Type: text/html
```

```
127.0.0.1 - - [02/Mar/2011:13:01:45 +0000] "HEAD / HTTP/1.0" 200 182 "-" "-"
```

```
---- Wed Mar 2 13:01:45 GMT 2011
---- Wed Mar 2 13:01:55 GMT 2011
Requesting /nofile...
[Wed Mar 02 13:01:55 2011] [error] [client 127.0.0.1] File does not exist: /home/steve/sgp/newweb/nofile
HTTP/1.1 404 Not Found
Date: Wed, 02 Mar 2011 13:01:55 GMT
Server: Apache/2.2.16 (Debian)
Vary: Accept-Encoding
Content-Length: 274
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /nofile was not found on this server.</p>
<hr>
<address>Apache/2.2.16 (Debian) Server at ksgp Port 80</address>
</body></html>
127.0.0.1 - - [02/Mar/2011:13:01:55 +0000] "GET /nofile HTTP/1.0" 404 477 "-" "-"
#
```

用于调试也很有用，因为 PHP 错误清楚地显示成 HEAD /请求结果的形式，并且错误 404 显示为 GET /nofile 的请求结果。类似地，我们也能立即看到成功标志(HEAD /请求的 200)。

13.5 od

od 表示 octal dump，但它的功能要远比八进制处理多得多。它可以用各种有用格式中的任意一种处理与表示数据。在第 14 章比较 factor 的 GNU 与非 GNU 实现时，我们使用

`od` 确认了原始(非 GNU)实现使用两个“空格”字符来填充结果。这两个空格被显示在两行，而 GNU 实现会将它们显示在一行。首先，如果我们观察输出，它看起来像是一对空格，但无法判断数字之后是否有空白字符或者到底填充的是什么的空白字符。将输出通过管道传递给 `od -a`，我们得到人能看明白的解释：6 后面是一个换行符、两个空格、2 与换行符，然后是两个空格、3 与最后一个换行符。

```
$ factor 6
6
 2
 3
$ factor 6 | od -a
0000000 6 nl sp sp 2 nl sp sp 3 nl
0000012
```

将相同的内容用管道传递给 `od -x`，我们得到十六进制格式的信息。因为系统使用的是低位在前(little-endian)架构，所以字节被反过来。0a36 表示的是两个字符，0x36 表示 6，0x0a 表示换行符。下一个双字节是两个 0x20 字符，它们是空格。然后，0x32 表示 2，0x0a 表示换行符。0x2020 是两个空格，然后 0x33 表示 3，0x0a 表示最后一个换行符。`ascii(7)`手册页包含所有 ASCII 字符的十进制、十六进制与八进制表示。

```
$ factor 6 | od -x
0000000 0a36 2020 0a32 2020 0a33
0000012
```

还有一个表示“惯例”的 `-t` 标志，它进一步使用 `-x1` 选项直接按照字节顺序进行十六进制格式化，而不是按照两字节的字来进行。这比 `od -x` 格式更容易阅读，尤其是在低位在前的硬件中。使用 `od` 的这些选项显示相同的输出，看看它们之间的区别。

```
$ factor 6
6
 2
 3
$ factor 6 | od -a
0000000 6 nl sp sp 2 nl sp sp 3 nl
0000012
$ factor 6 | od -x
0000000 0a36 2020 0a32 2020 0a33
0000012
$ factor 6 | od -t x1
0000000 36 0a 20 20 32 0a 20 20 33 0a
0000012
$
```

更实用的对数据进行比特级审查的一个例子是 x86 系统中磁盘的主引导记录(Master Boot Record, MBR)。如果 MBR 损坏，则机器无法启动。MBR 是位于磁盘开头的经过很好文档化的由 512 字节组成的结构。前 440 个字节是可执行代码(grub 或 Microsoft Windows

启动引导程序的第一部分)。接下来 6 个字节是磁盘签名与填充数据,所以前 446 个字节对分区表而言是没有用的数据。在这之后是 4 条 16 字节的记录,每条都表示一个主分区。最后的两个字节 0xaa55 标志着 MBR 的结束,也作为一个签名来确认前 510 个字节是真正的 MBR 而非随机数据。这对 BIOS 比较有用,因为在执行前 440 个字节的代码前要进行确认,而且也可以用在下面的代码中确定 MBR 的状态。

下面的脚本使用 -b 测试来检测传递的是设备(如/dev/sda)还是常规文件。这使得脚本可以通过使用 dd 提取前 512 个字节来查看运行中机器的磁盘。然而要使该脚本真正有用,它需要有能力查看已损坏系统的 MBR: 通过从 CD-ROM 或网络启动损坏的系统,我们可以提取它的 MBR,然后送到一个正常运行的机器上检查。可以不首先调用 dd 而以这种方式处理常规文件。

对于每个分区,第 1 个字节的值为 0x00(0 表示不可启动分区)或者 0x80(128 表示可启动分区)。其他任何字节都是非法的。第 5 个字节包含分区类型——Linux、FAT32、LVM、NTFS 与 swap 等。这条记录的余下部分包括分区的起始与终止位置(f1、f2、f3、l1、l2、l3)、第一个扇区的 LBA 以及分区中的扇区数目。该脚本使用第 0、4 字节以及第 13~16 字节分别显示可启动标志、分区类型和扇区大小。

使用 bc 与 printf 进行一些有趣的操作,将小写的十六进制扇区大小转换为十进制字节。该操作基于扇区大小为 512 字节,所以两个扇区是 1KB,再除以 1 000 两次得到 GB 数(在存储器行业中,1GB 是 1 000MB,不是 1 024MB)。

```
# cat mbr.sh
#!/bin/bash
if [ -b $1 ]; then
    mbr=`mktemp`
    echo "Reading MBR from device $1"
    dd if=$1 of=$mbr bs=512 count=1
    mbr_is_temporary=1
else
    mbr=$1
    if [ -r "$mbr" ]; then
        echo "Reading MBR from file $mbr"
    else
        echo "Readable MBR required."
        exit 1
    fi
fi

od -v -t x1 -An -j 510 $mbr |grep -q " 55 aa$"
if [ "$?" -ne "0" ]; then
    echo "MBR signature not found. Not a valid MBR."
    exit 1
fi

partnum=1
od -v -t x1 -An -j446 -N 64 $mbr | \
while read status f1 f2 f3 parttype l1 l2 l3 lba1 lba2 lba3 lba4 s1 s2 s3 s4
```

```

do
    if [ "$parttype" == "00" ]; then
        echo "Partition $partnum is not defined."
    else
        case $status in
            00) bootable="unbootable" ;;
            80) bootable="bootable" ;;
            *) bootable="invalid";
        esac
        printf "Partition %d is type %02s and is %s." $partnum $parttype $bootable
        sectors=`printf "%02s%02s%02s%02s\n" $s4 $s3 $s2 $s1 | \
            tr '[:lower:]' '[:upper:]'`
        bytes=`echo "ibase=16; $sectors / 2" | bc`
        gb=`echo "scale=2; $bytes / 1000 / 1000" | bc`
        printf " Size %.02f GB\n" $gb
    fi
    partnum=`expr $partnum + 1`
done
if [ "$mbr_is_temporary" ]; then
    rm -f $mbr
fi
# ./mbr.sh /dev/sda
Reading MBR from device /dev/sda
1+0 records in
1+0 records out
512 bytes (512 B) copied, 7.5848e-05 s, 6.8 MB/s
Partition 1 is type 07 and is bootable. Size 25.70 GB
Partition 2 is type bf and is bootable. Size 32.95 GB
Partition 3 is type 05 and is unbootable. Size 58.55 GB
Partition 4 is not defined.
# fdisk -l /dev/sda

Disk /dev/sda: 120.0 GB, 120034123776 bytes
255 heads, 63 sectors/track, 14593 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x8b213c0c

   Device Boot      Start         End      Blocks    Id  System
/dev/sda1  *           1          3200     25703968+    7  HPFS/NTFS
/dev/sda2  *        3201         7303     32957347+   bf  Solaris
/dev/sda3             7304        14593     58556925    5  Extended
/dev/sda5  *        7304        10342     24410736   83  Linux
/dev/sda6             10343        10585      1951866   82  Linux swap / Solaris
/dev/sda7             10586        14593     32194228+   83  Linux
# ./mbr.sh /tmp/suspect-mbr.bin
Reading MBR from file /tmp/suspect-mbr.bin
Partition 1 is type 27 and is unbootable. Size 12.58 GB
Partition 2 is type 07 and is bootable. Size 0.10 GB
Partition 3 is type 07 and is unbootable. Size 58.59 GB
Partition 4 is type 05 and is unbootable. Size 172.91 GB

```

Solaris x86 的 fdisk 分区表与 Linux 系统的一样。Linux 系统在每个 fdisk 分区上放置一个文件系统；而 Solaris x86 将 fdisk 分区作为虚拟磁盘(默认对该分区使用整个物理磁盘)，将虚拟磁盘分成很多片，并将每一片都配置成一个文件系统。表示第一个分区的特殊设备是 c0t0d0p0，且 MBR 可以在该设备的开头找到。这与 Linux 更精确的表示不同，因为 c0t0d0p0 与/dev/sda1 而不是/dev/sda 等价。

```
solarisx86# ./mbr.sh /dev/dsk/c0t0d0p0
Reading MBR from device /dev/dsk/c0t0d0p0
1+0 records in
1+0 records out
Partition 1 is type 82 and is bootable. Size 143.23 GB
Partition 2 is not defined.
Partition 3 is not defined.
Partition 4 is not defined.
$
```

13.6 paste

paste 命令与 **cut** 相反。它将多个文件粘贴到一起，(默认)使用制表符分开。不要与图形用户界面(GUI)中使用的术语(“剪切”与“粘贴”)混淆。**paste** 命令从多个文件中创建表格数据，所以如果给定 3 个文件，它会把它们用 3 列放在一起。第一列来自文件 1，中间一列来自文件 2，最后一列来自文件 3。它可以接受任意数目的输入文件。

下列脚本接收了一个输入文件 **hosts**，该文件包含一组主机。脚本使用该文件存储每个主机的 IP 地址与以太网地址。为确保如果 **grep** 或者 **getent** 命令失败||结构也会失败，我们要设置 **shell** 选项 **pipefail**。因此无论成功与否，脚本必定要向 **\$IPS** 与 **\$ETHERS** 中写入一行文本。这确保了当粘贴到一起时，文件能保持同步。如果设置了 **pipefail**，则管道中任意一处的失败都会导致整个管道的失败。默认情况下，管道的返回值是管道中最后一个命令的返回值。这意味着 **getent hosts | cut** 成功与否由 **cut** 的返回码决定，而 **cut** 几乎总是会返回 0 表示成功。

```
$ cat hosts
localhost
plug
declan
atomic
broken
goldie
elvis
$ cat /etc/ethers
0a:00:27:00:00:00 plug
01:00:3a:10:21:fe declan
71:1f:04:e3:1b:13 atomic
01:01:8d:07:3a:ea goldie
01:01:31:09:2a:f2 elvis
```

```
$ cat gethosts.sh
#!/bin/bash
HOSTS=hosts
ETHERS=ethers
IPS=ips

set -o pipefail
for host in `cat $HOSTS`
do
    echo -en "${host}..."
    getent hosts $host | cut -d" " -f1 >> $IPS || echo "missing" >> $IPS
    grep -w "${host}" /etc/ethers | cut -d" " -f2 >> $ETHERS \
        || echo "missing" >> $ETHERS
done
echo
paste $HOSTS $IPS $ETHERS
$ ./gethosts.sh
localhost...plug...declan...atomic...broken...goldie...elvis...
localhost      ::1      missing
plug   192.168.1.5      0a:00:27:00:00:00
declan 192.168.1.10     01:00:3a:10:21:fe
atomic  192.168.1.11    71:1f:04:e3:1b:13
broken  missing missing
goldie  192.168.1.13    01:01:8d:07:3a:ea
elvis   192.168.1.227   01:01:31:09:2a:f2
$
```

尽管定界符可以使用 **-d** 标志修改，但默认是 **TAB** 字符。**cut** 使用定界符中的第一个字符来标记第 1 列与第 2 列，使用第二个字符标记第 2 列与第 3 列，依此类推。如果列数比定界符多，则 **cut** 会循环使用之前的定界符。**-d** 选项使用 **CSV** 格式(Comma-Separated Values)的输出，可以用电子表格程序编辑。**-s** 选项一次粘贴一个文件，并且高效地从列向数据转到横向数据。



CSV 文件格式本质上是有缺陷的。它无法处理字段中的逗号，或者必须对包含逗号的字段使用引号。如果使用了引号，那么其他引号就要自行引用。*The Art of Unix Programming*(ISBN 0131429019; Eric S. Raymond, 2008)讨论了各种文本文件格式与它们的含义，见 <http://www.catb.org/~esr/writings/taoup/html/ch05s02.html>。Joel Spolsky 写了一篇较好的后续文章，见 <http://www.joelonsoftware.com/articles/Biculturalism.html>。


```
$ paste -d, hosts ips ethers > ip.csv
localhost,::1,missing
plug,192.168.1.5,0a:00:27:00:00:00
declan,192.168.1.10,01:00:3a:10:21:fe
atomic,192.168.1.11,71:1f:04:e3:1b:13
broken,missing,missing
```

```
goldie,192.168.1.13,01:01:8d:07:3a:ea
elvis,192.168.1.227,01:01:31:09:2a:f2
$ paste -d, -s hosts ips ethers >> ip.csv
$ oocalc ip.csv
```

该脚本的运行结果如图 13-1 所示。

	A	B	C	D	E	F	G
1	localhost	::1	missing	regular paste produces columns			
2	plug	192.168.1.5	0a:00:27:00:00:00				
3	declan	192.168.1.10	01:00:3a:10:21:fe				
4	atomic	192.168.1.11	71:1f:04:e3:1b:13				
5	broken	missing	missing				
6	goldie	192.168.1.13	01:01:8d:07:3a:ea				
7	elvis	192.168.1.227	01:01:31:09:2a:f2			paste -s spins the axis	
8	localhost	plug	declan	atomic	broken	goldie	elvis
9	::1	192.168.1.5	192.168.1.10	192.168.1.11	missing	192.168.1.13	192.168.1.227
10	missing	0a:00:27:00:00:00	01:00:3a:10:21:fe	71:1f:04:e3:1b:13	missing	01:01:31:09:2a:f2	01:01:8d:07:3a:ea
11							

图 13-1



paste -s 也能用于删除文件中的所有回车符号，因为它将第一个输入文件粘贴到单独的第一行，然后类似地将第二个输入文件粘贴到单独的第二行，依此类推。

给主机名加上括号能更好地进行演示。不过，必须用其他方式添加空格。

```
$ paste -d"()" : " ips hosts ethers
::1(localhost)missing
192.168.1.5(plug)0a:00:27:00:00:00
192.168.1.10(declan)01:00:3a:10:21:fe
192.168.1.11(atomic)71:1f:04:e3:1b:13
missing(broken)missing
192.168.1.13(goldie)01:01:8d:07:3a:ea
192.168.1.227(elvis)01:01:31:09:2a:f2
$
```

最后，借助于\n 字符，定界符可以用来将文本分组到固定的一些列中。\\n 表示换行，\\t 表示 TAB。所以，定界符“tab、tab、tab 与换行”或者说是\\t\\t\\t\\n 会将前 3 列用 tab 分开，然后在第 4 列之后加上一个换行符，将它们组成 4 列。下面这个例子使用管道连接一个产生数字 1~27 的 seq。整个命令对表示成-的 stdin 文件进行处理。

```
$ seq 1 27 | paste -s -d "\\t\\t\\t\\n" -
1      2      3      4
5      6      7      8
9      10     11     12
13     14     15     16
17     18     19     20
21     22     23     24
25     26     27
$
```

13.7 pr

`pr` 是另一个不太为人所知的格式化命令，但在适当的情况下非常有用。标题通常不用，所以可以用 `pr -t` 忽略标题。还可以使用 `-T` 忽略页码。页码在屏幕上显示比打印排版通常更有用。在 `worldcup.sh` 脚本中，`pr` 的作用有点像 `join`，它用来将 4 列文本粘贴到单独一页中。它还能将单个输入分割成多个列，这与交互运行时 `ls` 命令非常相似。本例中，一个 30 条河流的列表要一个接一个列出来可能太多，但 `pr` 可以按照需要自动将它们分割成很多个井然有序的列。

```
$ wc -l rivers.txt
30 rivers.txt
$ pr -T -4 rivers.txt
Alamance      Deschutes      Koochiching     Rappahannock
Asotin        Escambia       Latah           Scioto
Beaver        Greenbrier     Merrimack       Tallapoosa
Boise         Humboldt       Mississippi      Tensas
Brazos        Iowa           Neosho          Vermilion
Cassia        Judith         Basin           Osage Wabash
Chattahoochee Kalamazoo      Platte          Yamill
Clearwater    Kankakee
$ pr -T -5 rivers.txt
Alamance      Chattahoochee  Iowa           Merrimack       Scioto
Asotin        Clearwater     Judith Basin   Mississippi     Tallapoosa
Beaver        Deschutes      Kalamazoo      Neosho          Tensas
Boise         Escambia       Kankakee       Osage           Vermilion
Brazos        Greenbrier     Koochiching    Platte          Wabash
Cassia        Humboldt      Latah          Rappahannock    Yamill
$
```

如果列数过多，`pr` 会对文本进行修剪来对齐。如果需要的仅仅是列向输出而不需要将输出排成行，我们可以使用 `-J` 选项。在不对文本进行修剪的前提下，我们没有足够的空间将输出显示为 6 列。`pr` 可以使用 `-J` 将它们紧凑地排在一起。

```
$ pr -T -6 rivers.txt
Alamance      Cassia      Greenbrier     Kankakee      Neosho      Tallapoosa
Asotin        Chattahooch Humboldt       Koochiching    Osage        Tensas
Beaver        Clearwater  Iowa          Latah         Platte       Vermilion
Boise         Deschutes   Judith Basi    Merrimack     Rappahannoc Wabash
Brazos        Escambia    Kalamazoo     Mississippi    Scioto       Yamill
$ pr -TJ -6 rivers.txt
Alamance      Cassia      Greenbrier     Kankakee      Neosho      Tallapoosa
Asotin        Chattahoochee Humboldt       Koochiching    Osage        Tensas
Beaver        Clearwater  Iowa          Latah         Platte       Vermilion
Boise         Deschutes   Judith Basin   Merrimack     Rappahannock Wabash
Brazos        Escambia    Kalamazoo     Mississippi    Scioto       Yamill
$
```


13.8 printf

尽管 `echo` 可能更简单，但 `printf` 可能比它更有用。`printf` 的主要功能之一是可以处理 C 风格的文本填充。在显示列向数据或使用其他显示元素将数据排列整齐时，`printf` 很有用。例如，格式化字符串 `%-10s` 表示字符串左对齐，宽度至少为 10 个字符。

下面的脚本用简单的方式读取 `/etc/passwd` 中的条目。它使用 `IFS` 变量让 `read` 使用冒号作为定界符，然后将这些变量传递给 `printf`，使用适当的填充将它们显示出来。注意，`avahi-autoipd` 与 `Debian-exim` 账号包含的字符数大于 10，所以输出向右移动，且没有经过修剪。类似地，`gnats` 与 `timidity` 的 `GECOS`（“名称”）字段中字符数超过 30，所以它们的 `shell` 字段会向右移动。

```
$ cat printf1.sh
#!/bin/bash

printf "%-10s %-30s %-10s\n" "Username" "Name" "Shell"
cut -d: -f1,5,7 /etc/passwd | while IFS=: read uname name shell
do
    printf "%-10s %-30s %-10s\n" "$uname" "$name" "$shell"
done
$ ./printf1.sh
Username      Name                               Shell
root          root                               /bin/bash
daemon        daemon                             /bin/sh
bin           bin                                /bin/sh
sys           sys                                /bin/sh
sync          sync                               /bin/sync
games         games                              /bin/sh
man           man                                /bin/sh
lp            lp                                 /bin/sh
gnats         Gnats Bug-Reporting System (admin) /bin/sh
nobody        nobody                            /bin/sh
libuuid       /bin/sh
messagebus    /bin/false
avahi-autoipd Avahi autoip daemon,,,           /bin/false
festival      /bin/false
gdm           Gnome Display Manager            /bin/false
haldaemon     Hardware abstraction layer,,,     /bin/false
usbmux        usbmux daemon,,,                 /bin/false
sshd          /usr/sbin/nologin
saned         /bin/false
avahi         Avahi mDNS daemon,,,             /bin/false
ntp           /bin/false
Debian-exim   /bin/false
Timidity      TiMidity++ MIDI sequencer service /bin/false
$
```

`printf` 所能理解的不仅仅是变量的长度。它还可以用很好的方式对齐数值输出。下面的脚本使用 `echo` 显示 1~10 的平方，然后使用 `printf` 突出不同值之间的差异。

最后，脚本计算带 10 位小数位(`scale=10`)的 1~10 的平方根。然而，给 `printf` 输入的格式化字符串为 `%0.4f`，所以无论是否需要，还是显示 4 位小数位。

```
$ cat printf2.sh
#!/bin/bash
for i in `seq 1 10`
do
    echo "$i squared is `expr $i \* $i`"
done

for i in `seq 1 10`
do
    printf "%2d squared is %3d\n" $i `expr $i \* $i`
done

for i in `seq 1 10`
do
    printf "The square root of %2d is %0.4f\n" $i `echo "scale=10;sqrt($i)"|bc`
done
$ ./printf2.sh
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
The square root of 1 is 1.0000
The square root of 2 is 1.4142
The square root of 3 is 1.7321
The square root of 4 is 2.0000
The square root of 5 is 2.2361
The square root of 6 is 2.4495
The square root of 7 is 2.6458
```

```
The square root of 8 is 2.8284
The square root of 9 is 3.0000
The square root of 10 is 3.1623
```

13.9 shuf

shuf 是比较有用的排序器，一般用来使用随机输入产生随机乱序的输出，尽管我们可以用自行提供的“随机”源每次产生可重复的结果。它可以作用于输入文件或者数值范围。对数组进行操作也非常有用。

13.9.1 掷骰子

下面这个简单的脚本模拟了掷出 3 个骰子的动作，然后给出它们的总和。该脚本还演示了将这样的功能放到函数中是如何使得与脚本的整合更灵活与更容易的。不仅仅是代码的主体部分更加简单易懂，而且一旦对实现方式进行了修改，效果会传到脚本的其他部分。



可从
wrox.com
下载源代码

```
$ cat dice.sh
#!/bin/bash

# could even do this as an alias.
function rolldice
{
    return `shuf -i 1-6 -n1`
}

total=0
rolldice
roll=$?
total=`expr $total + $roll`
echo "First roll was $roll"

rolldice
roll=$?
total=`expr $total + $roll`
echo "Second roll was $roll"

rolldice
roll=$?
total=`expr $total + $roll`
echo "Third roll was $roll"

echo
echo "Total is $total"

$ ./dice.sh
First roll was 5
Second roll was 3
```

```
Third roll was 2
Total is 10
```

```
$ ./dice.sh
First roll was 3
Second roll was 4
Third roll was 5
```

```
Total is 12
$ ./dice.sh
First roll was 4
Second roll was 2
Third roll was 2
```

```
Total is 8
$
```

dice.sh

13.9.2 发牌

更高级一些的例程实现了随机发牌。这个脚本对两个数组使用 `shuf`，打乱扑克牌的花色和数字。随机性是这样实现的，先拒绝前3次抽到的牌，然后将第4次抽到的牌保存到变量中。



可从
wrox.com
下载源代码

```
$ cat cards.sh
#!/bin/bash

suits=( diamonds clubs hearts spades )
values=( one two three four five
          six seven eight nine ten
          jack queen king )

function randomcard
{
    echo "the `shuf -nl -e ${values[@]}` of `shuf -nl -e "${suits[@]}"`"
}

echo "You rejected `randomcard` and put it back in the deck."
echo "You rejected `randomcard` and put it back in the deck."
echo "You rejected `randomcard` and put it back in the deck."
YOURCARD=`randomcard`
echo "You picked $YOURCARD."
echo "I remember $YOURCARD so it is no longer random."
echo "It will always be $YOURCARD."

$ ./cards.sh
You rejected the jack of clubs and put it back in the deck.
You rejected the seven of diamonds and put it back in the deck.
You rejected the four of hearts and put it back in the deck.
You picked the three of spades.
```

```
I remember the three of spades so it is no longer random.
It will always be the three of spades.
$ ./cards.sh
You rejected the ten of hearts and put it back in the deck.
You rejected the two of diamonds and put it back in the deck.
You rejected the queen of spades and put it back in the deck.
You picked the six of diamonds.
I remember the six of diamonds so it is no longer random.
It will always be the six of diamonds.
$
```

cards.sh

因为我们可以自己定义随机源,所以可以对代码进行一些修改使随机性或多或少具有可预测性。对 `randomcard` 函数的修改意味着(在我使用的系统中)总是会抽到梅花 J。读者所使用的系统中的 `/etc/hosts` 不一样,所以结果和我的不同,但每次运行都会产生相同的结果。



可从
wrox.com
下载源代码

```
function randomcard
{
    echo "the `shuf --random-source=/etc/hosts -nl -e ${values[@]}` of\"
        "`shuf --random-source=/etc/hosts -nl -e "${suits[@]}"`"
}
$ ./cards-lessrandom.sh
You rejected the jack of clubs and put it back in the deck.
You rejected the jack of clubs and put it back in the deck.
You rejected the jack of clubs and put it back in the deck.
You picked the jack of clubs.
I remember the jack of clubs so it is no longer random.
It will always be the jack of clubs.
$
```

cards-lessrandom.sh

可以通过创建完全已知的随机源来产生完全可预测的序列。例如,使用 `23549yer0tirgogti435r4gt9df0gtire`(在键盘上随机敲入的字符)总是会抽到红桃 Q。将 `randomcard` 中的 `/etc/hosts` 替换为 `/tmp/random` 来重现这一点。

```
$ echo 23549yer0tirgogti435r4gt9df0gtire > /tmp/random
$ ./cards-lessrandom.sh
You rejected the queen of hearts and put it back in the deck.
You rejected the queen of hearts and put it back in the deck.
You rejected the queen of hearts and put it back in the deck.
You picked the queen of hearts.
I remember the queen of hearts so it is no longer random.
It will always be the queen of hearts.
$
```

13.9.3 旅行线路

最后介绍一个更复杂的脚本,它以惊人的一致性进行旅游线路的随机选择。当它建议

去纽约时，可能会推荐访问一下自由女神像。但当建议去悉尼时，它推荐去悉尼歌剧院，所以并不是完全随机的。该脚本使用一个子目录 `places/`，它包含了用已知目的地命名的文本文件。文件的每一行包含当地的一个旅游景点。`tourism.sh` 中的 `while` 循环确保最后一个景点之前没有单词 `and`。也就是说我们可以通过命令行修改旅游的天数，并且能保证英语语法的正确性。脚本还会在最后连接的一对景点之前使用逗号，并且能正确地显示一日游。



可从
wrox.com
下载源代码

```
$ ls -l places/
total 32
-rw-rw-r-- 1 steve steve 46 Feb 19 12:04 Amsterdam
-rw-rw-r-- 1 steve steve 38 Feb 18 18:36 London
-rw-rw-r-- 1 steve steve 65 Feb 19 12:08 Microsoft
-rw-rw-r-- 1 steve steve 61 Feb 19 12:06 New York
-rw-rw-r-- 1 steve steve 38 Feb 18 18:49 Paris
-rw-rw-r-- 1 steve steve 53 Feb 19 12:08 Seattle
-rw-rw-r-- 1 steve steve 72 Feb 19 12:03 Sydney
-rw-rw-r-- 1 steve steve 47 Feb 19 12:08 Unix
$ cat places/Paris
the Eiffel Tower
Notre Dame
the Seine
$ cat tourism.sh
#!/bin/bash
cd places
place=$(shuf -e -n1 *)
days=${1:-2}

echo -en "Let's go to $place and check out "
count=1
shuf -n$days "$place" | while read trip
do
    let count=$count+1
    echo -en $trip
    if [ "$count" -le "`expr $days - 1`" ]; then
        echo -en ", "
    elif [ "$count" -le "$days" ]; then
        echo -en " and "
    else
        echo " "
    fi
done
$ ./tourism.sh
Let's go to Sydney and check out the Opera House and Sydney Harbour Bridge
$ ./tourism.sh 1
Let's go to New York and check out the Empire State Building
$ ./tourism.sh 3
Let's go to Paris and check out Notre Dame, the Eiffel Tower and the Seine
$ ./tourism.sh 4
Let's go to London and check out the London Eye, the Houses of Parliament, Big Ben
and Covent Garden
```

`tourism.sh`

13.10 sort

sort 是 Unix/Linux 工具箱中一个非常强大的实用程序。它可以按照各种标准进行排序、可以检查与合并排序过的文件、可以按照不同的键进行排序，甚至可以在这些键中按照不同的字符排序。它还可以删除重复，这样就省去了 `sort file.txt | uniq` 语法。

sort 的开关分为两类。一类是修改 sort 行为的开关。`-u` 删除重复结果，`-c` 与 `-C` 只检查输入是否已经过排序。`-t` 指定空白字符以外的字段分隔符，`-s` 将相同的行按原输入的顺序放置。其他开关修改 sort 实际用来对输入进行排序的策略。表 13-1 对 sort 能用来修改排序策略的标志进行了归纳。

表 13-1 排序修改符

标 志	作 用
-M	按照月份排序，unknown < Jan < Dec(取决于区域设置)
-b	忽略前置空白字符
-d	字典排序，忽略标点符号
-f	不区分大小写的排序
-g	通用的数值排序，在大多数情况下使用-n
-i	忽略非打印字符
-h	按照人类可读的方式对文件大小排序
-n	数值排序，9 在 10 之前
-R	随机排序
-r	翻转排序结果
-V	使用一种排序算法，该算法将软件版本通常理解为 foo-1.23a.093 这种形式



sort 的 -M 功能取决于当前的区域设置，特别是 LC_TIME 变量。我们可以通过 `locale -a` 命令查看安装了哪些区域设置。

13.10.1 按照键进行排序

`-k` 开关告诉 sort 用作排序键的一个字段(或一些字段)。默认的定界符是任意数量的空白字符，所以对于像下面的 `musicians.txt` 这样的相当杂乱无章的输入文件，`sort -k 2` 命令可以对姓氏按字母表排序。实际上没有将字段调整整齐，但没有任何关系，sort 默认将任意数量的空白字符当成字段定界符。



`-k` 开关的格式在很多年前修改过。以前的格式很少见，所以现在的书籍都不提及。

```
$ cat musicians.txt
Freddie Mercury      Singer   Queen   5 Sep 1946
Brian May             Guitarist Queen  19 Jul 1947
John Deacon           Bass     Queen   19 Aug 1951
Roger Taylor          Drums    Queen   26 Jul 1949
Benny Andersson       Pianist  Abba     16 Dec 1946
Bjorn Ulvaeus          Guitarist Abba     25 Apr 1945
Anni-Frid Lyngstad    Singer   Abba     15 Nov 1945
Agnetha Faltskog      Singer   Abba     5 Apr 1950
$ sort -k2 musicians.txt
Benny Andersson       Pianist  Abba     16 Dec 1946
John Deacon           Bass     Queen    19 Aug 1951
Agnetha Faltskog      Singer   Abba     5 Apr 1950
Anni-Frid Lyngstad    Singer   Abba     15 Nov 1945
Brian May             Guitarist Queen   19 Jul 1947
Freddie Mercury       Singer   Queen    5 Sep 1946
Roger Taylor          Drums    Queen    26 Jul 1949
Bjorn Ulvaeus          Guitarist Abba     25 Apr 1945
$
```

对多个字段进行排序也是可能的。要按照音乐家的年龄进行安排，可以按照出生的年、月、日进行排序。按照所需的顺序指定键，然后指定每个键的排序方式。日与年都是数值形式，所以使用表 13-1 中的 `-n` 开关，而 `-M` 表示按月份的三字母缩写形式排序。

```
$ sort -k7n -k6M -k5n musicians.txt
Bjorn Ulvaeus          Guitarist Abba     25 Apr 1945
Anni-Frid Lyngstad     Singer    Abba     15 Nov 1945
Freddie Mercury        Singer    Queen    5 Sep 1946
Benny Andersson        Pianist   Abba     16 Dec 1946
Brian May              Guitarist Queen    19 Jul 1947
Roger Taylor           Drums     Queen    26 Jul 1949
Agnetha Faltskog        Singer    Abba     5 Apr 1950
John Deacon            Bass       Queen    19 Aug 1951
$
```

更实用的例子是下面对 `/etc/hosts` 文件的排序。按照主机名排序非常有用。通过 IP 地址按数值顺序对文件进行排序则更不错。这有些复杂，IP 地址的定界符是点号，所以 `-t` 开关可以用来指定定界符。数据可以先使用键 1 排序，然后是键 2、键 3，最后是键 4。这有些不太直观，因为 `sort -t. -k1 -k2 -k3 -k4` 不会按照预期的方式排序。键 1 被当成初始的“127.”，然后是一行剩下的部分。所以到键 4 后，命令会用 IP 地址的最后一个八位字节(.1、.210、.227 等)进行重新排序，之前的 3 个排序完全没有起到作用。

```
$ cat hosts
127.0.0.1      localhost
192.168.1.3    sky
192.168.1.11   atomic
192.168.1.10   declan        declan.example.com
192.168.0.210  dgoldie ksgp dalston
```



```

192.168.1.5      plug
192.168.1.227   elvis
192.168.1.13    goldie goldie.example.com smf spo sgp
$ sort -t. -k1 -k2 -k3 -k4 hosts
127.0.0.1       localhost
192.168.0.210   dgoldie ksgp dalston
192.168.1.10    declan      declan.example.com
192.168.1.11    atomic
192.168.1.13    goldie goldie.example.com smf spo sgp
192.168.1.227   elvis
192.168.1.3     sky
192.168.1.5     plug

```

为了避免这一点，我们可以告诉 `sort` 每个键的起始字段与终止字段。在本例中，第一个键是只针对 1 的字段 1，第二个键是只针对 2 的字段 2，依此类推。这可以得到想要的结果，即 IP 地址被归到各自的子网中，并经过了数值排序。`-n` 标志同样告诉 `sort` 进行数值排序(不会进行字母表排序，所以 5 在 10 之前)。

```

$ sort -t. -k1,1n -k2,2n -k3,3n -k4,4n hosts
127.0.0.1       localhost
192.168.0.210   dgoldie ksgp dalston
192.168.1.3     sky
192.168.1.5     plug
192.168.1.10    declan      declan.example.com
192.168.1.11    atomic
192.168.1.13    goldie goldie.example.com smf spo sgp
192.168.1.227   elvis
$

```

13.10.2 按照日期与时间对日志文件排序

通过只对字段中某些字符进行排序，我们可以进一步对字段进行分解。标准的 Apache 访问日志文件将访问的日期与时间存储为[02/Mar/2011:13:06:17-0800]形式。要自动排序不是太容易，其中第一个键是 2011，第二个键是 Mar，然后是 02。之后分别是 13、06 与 17。除了序列(2011/Mar/02/13:06:17 会更简单一些)中的值以外，表示三月的 Mar 在 Apr 之前，但如果在此采用没有意义的字母表排序，则会在 Dec 之后。

幸运的是，正如在本节开头部分的音乐家示例中所看到的，`sort` 可以根据区域设置对月份进行排序。只要当前的区域设置使用与日志文件中相同的表示月份的名字，`sort` 就可以使用 `-M` 标志按照日历(从 Jan 到 Dec)顺序对月份进行排序。为了演示如何进行排序，我们给出这个典型的检查 3 个日志文件的系统管理员任务。我们讨论的网页(文章 928，由 URL 中的 `art=928` 可以看出)出现在两个较早的日志文件中，而不是当前的 `access_log`。然而，简单地使用 `grep` 只会按照“错误”的顺序处理这些文件。因为按照字母表顺序，`access_log.processed.1` 排在较早的 `access_log.processed.2` 之前并且文件名通配符扩展也会对结果进行字母表排序。这意味着日志不会按照希望的顺序显示。

```
$ ls -ltr access*
-rw-rw-r-- 1 steve steve 22429808 Mar 2 04:42 access_log.processed.2
-rw-rw-r-- 1 steve steve 45490104 Mar 4 04:49 access_log.processed.1
-rw-rw-r-- 1 steve steve 19457016 Mar 5 21:16 access_log
$ grep art=928 *
access_log.processed.1:77.88.31.247 -- [02/Mar/2011:13:06:17 -0800] "GET /urandom/
comment.php?art=928 HTTP/1.1" 200 11551 "-" "Mozilla/5.0 (compatible; YandexBot/3.0
; +http://yandex.com/bots)"
access_log.processed.1:77.88.31.247 -- [02/Mar/2011:16:30:34 -0800] "GET /urandom/
comment.php?title=Number+of+the+day&art=928 HTTP/1.1" 200 11599 "-" "Mozilla/5.0 (c
ompatible; YandexBot/3.0; +http://yandex.com/bots)"
access_log.processed.1:66.249.69.52 -- [04/Mar/2011:01:18:32 -0800] "GET /urandom/
comment.php?art=928 HTTP/1.1" 200 11584 "-" "Mozilla/5.0 (compatible; Googlebot/2.1
; +http://www.google.com/bot.html)"
access_log.processed.2:67.195.111.173 -- [01/Mar/2011:06:26:32 -0800] "GET /urando
m/comment.php?title=Number+of+the+day&art=928 HTTP/1.0" 200 11599 "-" "Mozilla/5.0
(compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"
access_log.processed.2:218.213.130.168 -- [01/Mar/2011:07:25:41 -0800] "GET /urand
om/comment.php?art=928 HTTP/1.1" 200 11551 "-" "ichiro/4.0 (http://help.goo.ne.jp/d
oor/crawler.html)"
access_log.processed.2:218.213.130.168 -- [01/Mar/2011:07:33:54 -0800] "GET /urand
om/comment.php?title=Number+of+the+day&art=928 HTTP/1.1" 200 11599 "-" "ichiro/4.0
(http://help.goo.ne.jp/door/crawler.html)"
```

在这个实例中，因为文件的时间戳是正确的，所以管理员可以使用 `ls -lt` 将文件以正确顺序排序。具体命令是 `grep art=928 `ls -tr access_log*``。但实际上日志可以是远程服务器上下载下来的并且都有相同的时间戳，或者它们的文件名不是按照某种逻辑顺序排列的。然而，下面的代码给出了按照日期排列这些文件的方法。代码对默认的 `sort` 语法进行了一些调整。排序类型本身可能是表 13-1 中的任意一种，并且可以用于字段的开头和/或结尾。4.10n,4.13n 与 4.10n,4.13 或 4.10,4.13n 是等价的。

在下面的例子中，`sort -k 4.10,4.13n` 告诉 `sort` 第一个最先排序的字段是年份，它包含在第 4 个字段的第 10~13 个字符中，并且让 `sort` 进行数值排序。第二个 `-k` 标志 `-k 4.6,4.8M` 告诉 `sort` 将第 4 个字段的第 6~8 个字符当成当前区域设置中的三字母月份缩写。剩下的部分在上一节介绍过。这样全部合起来就是年、月、日、时、分、秒。这可以按照要求的顺序产生结果：

```
$ grep art=928 * | sort -k 4.10,4.13n -k 4.6,4.8M -k 4.3,4.4n\
>      -k 4.15,4.16n -k 4.18,4.19n -k 4.21,4.22n
access_log.processed.2:67.195.111.173 -- [01/Mar/2011:06:26:32 -0800] "GET /urando
m/comment.php?title=Number+of+the+day&art=928 HTTP/1.0" 200 11599 "-" "Mozilla/5.0
(compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"
access_log.processed.2:218.213.130.168 -- [01/Mar/2011:07:25:41 -0800] "GET /urand
om/comment.php?art=928 HTTP/1.1" 200 11551 "-" "ichiro/4.0 (http://help.goo.ne.jp/d
oor/crawler.html)"
access_log.processed.2:218.213.130.168 -- [01/Mar/2011:07:33:54 -0800] "GET /urand
om/comment.php?title=Number+of+the+day&art=928 HTTP/1.1" 200 11599 "-" "ichiro/4.0
(http://help.goo.ne.jp/door/crawler.html)"
```

```
access_log.processed.1:77.88.31.247 -- [02/Mar/2011:13:06:17 -0800] "GET /urandom/
comment.php?art=928 HTTP/1.1" 200 11551 "-" "Mozilla/5.0 (compatible; YandexBot/3.0
; +http://yandex.com/bots)"
access_log.processed.1:77.88.31.247 -- [02/Mar/2011:16:30:34 -0800] "GET /urandom/
comment.php?title=Number+of+the+day&art=928 HTTP/1.1" 200 11599 "-" "Mozilla/5.0 (c
ompatible; YandexBot/3.0; +http://yandex.com/bots)"
access_log.processed.1:66.249.69.52 -- [04/Mar/2011:01:18:32 -0800] "GET /urandom/
comment.php?art=928 HTTP/1.1" 200 11584 "-" "Mozilla/5.0 (compatible; Googlebot/2.1
; +http://www.google.com/bot.html)"
$ grep art=928 `ls -tr access_log*`
```

计算字符串中字符数的较为方便的方式是在对象文本的上方或下方直接 `echo` 出一行数字。这一技术使得我们不用拿手指对着屏幕计算字符的数目。

```
$ echo " [02/Mar/2011:16:30:34 -0800]" ; echo "123456789012345678901234567890"
[02/Mar/2011:16:30:34 -0800]
123456789012345678901234567890
$
```



字段中的字符从字段开始前的第一个空白字符开始，所以 `Mar` 是字段中的第 6~8 个字符(不是通常所想的第 5~7 个)。`-b` 标志可以用来关闭这种行为方式。

13.10.3 对人类可读的数值进行排序

很多现代的实用程序可以很方便地将文件大小显示为人类可读的形式。2.6GB 就比 2751758391 字节要易懂。然而直到 2010 年的夏天，`sort` 才具备了分析这些不同形式的功能。2.6GB 明显要比 3.0MB 大，但 `sort` 必须理解单位后才能明白它们的大小关系。下面这个简短的脚本给出了某个目录中最大文件的前 10 名列表。脚本使用 `du -sh *` 给出未排序的结果，然后将它们以反向数值顺序排序。这使得最大的文件位于顶端，最小的文件位于底端。将结果通过管道送给 `head` 可以得到前 10 个(如果目录中至少有 10 个文件)，然后将它们用管道发送给 `cat -n` 加上前置的排名。这在 `/home` 中运行特别有用，它能看出哪个用户占用了最多的磁盘。



可从
wtox.com
下载源代码

```
# cat dirsized.sh
#!/bin/bash

cd "${1:-.}"
if [ "$?" -ne "0" ]; then
    echo "Error: Failed to change to directory $1"
    exit 2
fi
echo "The largest files/directories in $1 are:"
du -sh * | sort -hr | head | cat -n -
# ./dirsized.sh /var/log
The largest files/directories in /var/log are:
```

```

1  1.3M  installer
2  1.1M  kern.log.1
3  780K  messages.1
4  696K  apache2
5  680K  wtmp.1
6  572K  kern.log
7  412K  messages
8  380K  daemon.log.1
9  288K  syslog.1
10 284K  daemon.log
#

```

```
dirsize.sh
```

13.11 tr

`tr` 实用程序将单个字符转换成其他字符。如果调用 `echo frabacbable | tr 'a' 'b'`, 该命令会将字母 `a` 全都替换为 `b`, 结果输出为 `frbbcbbbble`。下面的脚本按照厂商安装说明的要求对内核进行调整。具体细节会随版本变化, 所以每次必须从厂商下载, 但有两件事是保持不变的:

- 厂商总是提供内核可调项, 就像 `echo value > /proc` 这样的临时修改, 而不是对 `sysctl.conf` 文件的修改。
- 厂商的排版很糟糕, 其中有非法的斜线与大写字母。必须将它们处理好。

有两种方法可以调整 Linux 内核。我们可以通过 `echo 65536 > /proc/sys/fs/file-max` 设置用户可以打开的文件的最大数目, 修改会立即发生, 但重启后不会保存; 或者可以在 `/etc/sysctl.conf` 中添加一行 `sys.fs.file-max = 65536`, 只要系统启动就会生效。我们将需要运行 `sysctl -p` 来动态地载入 `/etc/sysctl.conf` 中的新值。注意, `sys.fs.file-max` 在 `/proc` 文件系统中表示为 `sys/fs/file-max`。这对 `tr` 来说是非常理想的。我们有一个已知的只用来作为定界符的单个字符, 并且目标是将其替换为其他单个字符, 而且也只作为定界符。

`tr` 的另一个用法是将一定范围内的字符置换到另一个范围中的相对位置。它通常用来将 `[A-Z]` 置换为 `[a-z]`, 但其实可以置换任意字符。它还可以使用预定义的范围, 如 `[:alnum:]` 表示所有字母与数字、`[:digit:]` 表示数字、`[:space:]` 表示空白字符等。它们在 `tr(1)` 手册页中都有介绍。

下面的脚本接受当前版本注释(`relnotes.txt`), 提取所有 `echo value > /proc/tunable.subsys.subval` 命令, 然后将它们转换为适当的用于 `/etc/sysctl.conf` 的 `tunable.subsys.subval = value` 项。脚本使用 `tr '[A-Z]' '[a-z]'` 将大写字母转换为小写形式, 因为所有的内核可调项都是小写。脚本通过 `tr '/' '.'` 将斜线转换为点号, 还将任意重复的斜线删掉, 所以输入 `/pROc/sYs//kERnel//sHMMMax` 就变成了格式合法的输出 `sys.kernel.shmmax`。



可从
wrox.com
下载源代码

```

$ ls
installproduct.sh
$ cat installproduct.sh
#!/bin/bash

```

```

VERSION=${1:-"2.4.3"}
DOWNLOAD=downloads.vendor.com
URL=http://${DOWNLOAD}/product/v${VERSION}

echo "Retrieving release notes..."
wget -nd ${URL}/relnotes.txt

echo "Got release notes:"
echo "*** START OF RELEASE NOTES"
cat relnotes.txt
echo "*** END OF RELEASE NOTES"

grep "^echo " relnotes.txt | tr -s "/" | while read ignoreecho params
do
    value=`echo $params | cut -d">" -f1`
    proc=`echo $params | cut -d">" -f2 | cut -d"/" -f3-`
    sysctl=`echo $proc | tr '[A-Z]' '[a-z]' | tr '/' '.'`
    echo "Setting $sysctl to $value..."
    echo $sysctl = $value | tee -a /etc/sysctl.conf
done
echo "Loading new kernel values."
sysctl -p >/dev/null 2>&1
$ ./installproduct.sh
Retrieving release notes...
--2011-03-07 17:19:08-- http://downloads.vendor.com/product/v2.4.3/relnotes.txt
Resolving downloads.vendor.com... 192.168.1.13
Connecting to downloads.vendor.com|192.168.1.13|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 292 [text/plain]
Saving to: `relnotes.txt'

100%[=====>] 292    --.-K/s    in 0s

2011-03-07 17:19:08 (21.9 MB/s) - `relnotes.txt' saved [292/292]

Got release notes:
*** START OF RELEASE NOTES
Before installiNg This product, TUnE the kernel as follows:

echo 65536 > ///Proc//sys/FS/file-max
echo 2097152 > /prOC//sYs/kErnEl/SHmall
echo 2147483648 > /pROc/sYs///kERnel//sHMMMax
echo 4096 > /prOc/syS/kerNel//shmMni
echo 250 32000 100 128 > /prOc//sYS/KERNEL/SEm

ThEn run thE instAller ROUTine.
*** END OF RELEASE NOTES
Setting sys.fs.file-max to 65536 ...
sys.fs.file-max = 65536
Setting sys.kernel.shmall to 2097152 ...

```

```

sys.kernel.shmall = 2097152
Setting sys.kernel.shmmax to 2147483648 ...
sys.kernel.shmmax = 2147483648
Setting sys.kernel.shmmni to 4096 ...
sys.kernel.shmmni = 4096
Setting sys.kernel.sem to 250 32000 100 128 ...
sys.kernel.sem = 250 32000 100 128
$

```

[installproduct.sh](#)

如上文提到的，`tr` 功能不只是大小写转换。使用非对称集合，可以对指定字符进行压缩用于更严格的需求。来自某个网站的 `access_log` 包含了很多相关数据，但分析起来不容易。在下面的例子中，`cut` 语句只从日志中删除了 URL 请求：

```

$ cut -d' ' -f2 access_log
GET /forum/viewforum.php?f=2 HTTP/1.1
GET /forum/templates/subSilver/images/folder_lock_new.gif HTTP/1.1
GET /forum/templates/subSilver/images/folder_lock.gif HTTP/1.1
GET /forum/viewtopic.php?p=884&sid=a2738b9fc491726ac290aa7a9447291b HTTP/1.1
GET /forum/viewforum.php?f=2&sid=a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
GET /forum/profile.php?mode=register&sid=a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
GET /forum/viewtopic.php?p=840&sid=88c87f962c9ee6bdfff8b5fba9c728a8 HTTP/1.1
GET /forum/viewtopic.php?p=848 HTTP/1.1

```

如果`?`、`&`与`=`符号被删除，那么解释起来会容易得多。下面这个简单的 `tr` 语句将这样的字符替换为空格，使得很容易将文本传递给 `read` 语句：

```

$ cut -d' ' -f2 access_log | tr '[@&=]' ' '
GET /forum/viewforum.php f 2 HTTP/1.1
GET /forum/templates/subSilver/images/folder_lock_new.gif HTTP/1.1
GET /forum/templates/subSilver/images/folder_lock.gif HTTP/1.1
GET /forum/viewtopic.php p 884 sid a2738b9fc491726ac290aa7a9447291b HTTP/1.1
GET /forum/viewforum.php f 2 sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
GET /forum/profile.php mode register sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
GET /forum/viewtopic.php p 840 sid 88c87f962c9ee6bdfff8b5fba9c728a8 HTTP/1.1
GET /forum/viewtopic.php p 848 HTTP/1.1
$ cut -d' ' -f2 access_log | tr '[@&=]' ' ' | while read METHOD PAGE ARGUMENTS
> do
>   echo Page requested was $PAGE
>   echo Arguments were $ARGUMENTS
> done
Page requested was /forum/viewforum.php
Arguments were f 2 HTTP/1.1
Page requested was /forum/templates/subSilver/images/folder_lock_new.gif
Arguments were HTTP/1.1
Page requested was /forum/templates/subSilver/images/folder_lock.gif
Arguments were HTTP/1.1
Page requested was /forum/viewtopic.php
Arguments were p 884 sid a2738b9fc491726ac290aa7a9447291b HTTP/1.1

```

```

Page requested was /forum/viewforum.php
Arguments were f 2 sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
Page requested was /forum/profile.php
Arguments were mode register sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
Page requested was /forum/profile.php
Arguments were mode register agreed true sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
Page requested was /forum/posting.php
Arguments were mode newtopic f 2 sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
Page requested was /forum/login.php
Arguments were redirect posting.php mode newtopic f 2 sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
Page requested was /forum/posting.php
Arguments were mode newtopic f 2 sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
Page requested was /forum/login.php
Arguments were redirect posting.php mode newtopic f 2 sid a705161ccdf318a111c67dcde0e1bd03 HTTP/1.0
Page requested was /forum/viewtopic.php
Arguments were p 840 sid 88c87f962c9ee6bdf8b5fba9c728a8 HTTP/1.1
Page requested was /forum/viewtopic.php
Arguments were p 848 HTTP/1.1

```

另一个有用但不常见的 `tr` 开关是 `-d`。它可以删除任意由它给定的字符。该开关的众多功能之一是将以太网(MAC)地址从由冒号隔开的十六进制字节格式(70:5a:b6:2a:e8:b8)转换为也比较常用的无冒号格式(705ab62ae8b8)。使用 `tr -d` 完成这样的任务再合适不过。

```

$ ifconfig -a | grep HW
eth0      Link encap:Ethernet HWaddr 70:5a:b6:2a:e8:b8
pan0      Link encap:Ethernet HWaddr 6e:b4:bc:3a:bd:2e
vboxnet0  Link encap:Ethernet HWaddr 0a:00:27:00:00:00
wlan0     Link encap:Ethernet HWaddr 70:1a:04:e3:1b:10
$ ifconfig -a | grep HW | tr -d ':' | cut -c1-10,38-
eth0      705ab62ae8b8
pan0      6eb4bc3abd2e
vboxnet0  0a0027000000
wlan0     701a04e31b10$
$

```

13.12 uniq

`uniq` 初看起来像是只显示文件中唯一行的工具。实际上，它删除的是连续的重复行，但如果相同的行在输入中稍后出现，则还是会显示出来。因此，常见的组合是 `sort | uniq`。其实可以使用 `sort -u` 命令更高效地实现这一功能，因为没有必要产生第二个进程，然后在它们之间建立管道。

`uniq` 的用途是什么？这似乎问到点上了，而答案是 `uniq` 实际上在功能方面非常灵活。在第 12 章中，`uniq` 使用 `-w` 与 `-d` 标志只查找校验和相同的项。`-w` 标志告诉 `uniq` 只分析前

32 个字符(它们是校验和, 其余部分肯定是不同的), 而 **-d** 标志(如果输入数据已排序)只列出非唯一项。这个过滤结果的方法非常有用, 而在 **shell** 脚本的 **for** 循环或 **while** 循环中可能会比较累赘且耗时。



uniq 的一个相当独特的特性是, 如果在命令行中传递第二个文件名, 它会将输出写入到该文件而非 **stdout** 中去, 如果文件存在则将其覆盖。

与 **-d** 相反的是 **-u**。**uniq** 通常每次只输出一行, 而如果使用 **-u**, 它会只输出唯一行。如果有一项出现了两次, 则它根本不会被显示出来。3 种调用——单独的 **uniq**、**uniq -u** 与 **uniq -d**——为针对任意目的过滤任意输入提供了非常灵活的方法。其他主要过滤器用来忽略字符以及定义要进行比较的字段。

如第 12 章所介绍的, **-w32** 只会比较前 32 个字符。与 **-w N** 相反的是 **-s N**, 后者会忽略前 *N* 个字符。还有一个 **-f N** 标志, 它会忽略前 *N* 个字段, 其中字段使用空格和/或 **Tab** 分开。使用 **-i** 标志可以进行不区分大小写的文本比较。**uniq** 也可以给出每个输出行的数目; 使用 **-uc** 将总是 1, 使用 **-dc** 将总是大于 1。

以上这些使得 **uniq** 成为一个通常看起来像是比较过时、简单、目的单一的工具, 但实际上它非常灵活。我们在使用 **uniq** 的额外代码示例中演示了还能让 **uniq** 做些什么工作。很多人经常编写复杂而笨拙的脚本, 而实际上可以通过正确地使用 **uniq** 使这些脚本更精简且速度更快。

在一个唯一最低竞价拍卖会中, 赢得拍卖的不是出价最高的投标者, 而是出唯一最低价的投标者。这样一种场合下最适合使用 **uniq** 的 **-u** 标志。首先对竞价进行数值排序, **uniq -u -f1** 忽略第一个字段(竞标者姓名), 然后去掉非唯一竞价, 剩下的只是唯一的竞价。在下面的例子中, **Richard**、**Angela** 与 **Fred** 的竞价是唯一的。因为数据在送给 **uniq** 之前已经过排序, 所以第一条记录肯定是唯一的最低竞价。这个唯一的最低竞价刚开始是个不太好明白的概念, 但由于使用了 **uniq**, 我们使用一行简单的命令就能将其实现:

```
$ cat bidders
Dave 1.39
Bob 2.31
Albert 0.91
Elizabeth 1.39
Angela 1.09
Fred 3.13
Caroline 2.31
Rodger 0.91
Richard 0.98
$ sort -k2 -n bidders | uniq -u -f1 | head -1
Richard 0.98
$
```


13.13 wc

`wc` 表示 Word Count，但它也能统计字符与行数。这使得它能灵活地用于统计任意类型的数据。最常见的用法是统计文件或者(与大多数 Unix 工具一样)发送给它的其他任意数据中的行数，但也能统计字符与单词。

尽管通常只对一个文件使用，或者通过管道分析标准输入，但 `wc` 也能一次统计多个文件。`wc` 的 3 个主要标志是 `-w`(统计单词)、`-c`(统计字符)与 `-l`(统计行数)。其中最常用的是统计行数。统计文件行数经常要用到。统计管道中的结果也很常用。本书的很多代码与实用脚本都使用 `wc -l` 来自动对结果进行统计。

`wc` 的各种实现在处理多个文件时都会对输出进行填充，使得输出列更美观。在编写脚本时可能比较麻烦，因为“14”要解释成数字不如简单的“14”那么容易。`wc` 的 Unix 实现总是会进行填充，所以需要一些对应措施。`awk` 很容易用来删除空格，所以下面的命令使用 `awk` 可以正常工作。下面的命令显示了使用填充的多个文件，以及对将文件长度赋值给变量这样的常见任务的影响。

```
$ wc -l /etc/hosts*
 18 /etc/hosts
 14 /etc/hosts.allow
 87 /etc/hosts.deny
119 total
$ wc -l /etc/hosts
 18 /etc/hosts
$ num_hosts=`wc -l /etc/hosts | cut -d' ' -f1`
$ echo $num_hosts
18 /etc/hosts
$ num_hosts=`wc -l /etc/hosts | awk '{ print $1 }'`
$ echo $num_hosts
18
$
```

如果只有 `stdin` 或单个文件，`wc` 的 GNU 实现不会添加任何填充字符。这意味着如果只有一个文件，那么第一个较原始的从 `wc` 输出直接赋值给 `num_hosts` 的想法在 GNU 下能有效。在 `wc` 的其他实现中，需要使用 `awk` 获取实际的数值，而忽略填充字符。

在本章前面的世界杯脚本中，`NUMTEAMS` 变量是用这种方式定义的。如果要编写可移植的脚本，最安全的做法是假设 `wc` 的输出没有被安全地填充。如果脚本运行在相对较现代的机器中，而且不是处于运算密集型的循环中，这种额外的开销非常小。

```
NUMTEAMS=`wc -l $TEAMS | awk '{ print $1 }'`
```

本例没有造成太大的开销，并且确保了脚本更好的跨平台可移植性。因此，在分析 `wc` 的输出时最好使用这样的方法，除非能确定脚本中的 `wc` 是 GNU 实现且作用于单个输入。在这种情况下，我们可以使用轻量级的 `cut` 工具，如下所示：

```
NUMTEAMS=`wc -l $TEAMS | cut -d' ' -f1`
```

13.14 本章小结

Unix 与 Linux 在文本处理方面非常灵活。大多数配置都以文本格式存储，并且与工具无关，因此任何工具都可以用来对它们进行处理。这为用户与系统管理员提供了强大的功能，因为他们不再被限定使用某个工具编辑 `/etc/hosts`，而又要用另一个定制的工具编辑 `/etc/passwd`。相反的是，整个工具集可以用来处理几乎任何东西，并且这些工具可以按照各种不同的方式组合起来。这种清晰的设计是要让一切都开放且可访问。它是 Unix 模型的一个关键优势。

第 14 章将以本章和前一章为基础介绍一些实用的系统管理任务。

第 14 章

系统管理工具

系统管理是 shell 脚本编程的最常见任务。Unix 与 Linux 中有很多命令用来配置系统本身，所以大多数脚本都是用于这一目的也就不奇怪了。本章的目的在于给出一些如何使用这些命令的示例，特别是如何在 shell 脚本中高效地用它们自动化、扩展与简化系统管理任务。本章还讨论了与这些工具有关的陷阱与问题，以及这些工具在现实生活中的实际用法。本章是本书余下章节的基础。后者由一些全面而深入的实用脚本构成，更多地集中在这些脚本实际的功能，而不是它们使用的工具本身。

14.1 basename

basename 将目录路径去掉，返回文件的实际文件名。它最常用于在脚本中从 \$0 变量中提取脚本的名称。这可以用于调试与一般的消息输出。这种确定调用程序名称的功能也被一些系统实用程序用来修改其实际行为。**mount** 与 **umount** 命令共享了很多公共代码，但运行 **umount /home** 与运行 **mount /home** 的效果非常不同——而且是完全相反。单个程序通过检查调用程序名称并进行正确的操作来处理这种情况。这可以去除代码冗余，而将所有的系统挂载与卸载代码放在同一个程序中。一个简单的例子是后面将介绍的 **dos2unix** 转换实用程序。DOS(与 Microsoft Windows)文本文件格式使用 CR+LF，而 Unix(与 Linux)只使用 LF。这一区别很早就有了，并且永远也不太可能消除。有两个转换实用程序(**unix2dos** 与 **dos2unix**)用来实现两种文本文件格式的简单转换。它们不总是能在每个平台上找到，并且还有各种怪癖会影响到这两种非常类似的格式之间很简单的相互转换，但 **sed** 是实现这种转换的极佳工具。<http://sed.sourceforge.net/sed1line.txt> 网页中给出了很多有用的单行 **sed** 命令，其中有一些就用在了下面的脚本中。



tofrodos 包提供了 **fromdos** 和 **todos** 命令，是实现转换的另一种方式。

第一条命令 `ls -il` 显示 `dos2unix` 与 `unix2dos` 是硬链接。它们是具有两个目录项与不同名称的相同文件。`diff` 可以确认这一点。然而根据调用文件的不同,脚本会产生不同的行为。



```
$ ls -il dos2unix unix2dos
5161177 -rwxr-xr-x 2 steve steve 613 Feb 21 14:55 dos2unix
5161177 -rwxr-xr-x 2 steve steve 613 Feb 21 14:55 unix2dos
$ diff dos2unix unix2dos
$ echo $?
0
$ cat dos2unix
#!/bin/bash
# from http://sed.sf.net/sedlline.txt:
# sed 's/.$//'          # assumes that all lines end with CR/LF
# sed 's/$'"/`echo \\r`/" # command line under bash

if [ ! -f "$1" ]; then
    echo "Usage: `basename $0` filename"
    echo " `basename $0` converts between DOS and UNIX formats."
    echo " When called as unix2dos, it converts to DOS format."
    echo " Otherwise, it converts to UNIX format."
    exit 1
fi

case `basename $0` in
    unix2dos)
        sed -i 's/$'"/`echo \\r`/" $1
        exit $?
        ;;
    *) # Default to being dos2unix
        sed -i 's/.$//' $1
        exit $?
        ;;
esac
exit 0
```

dos2unix (与 unix2dos 一样的文件)

`cat -v` 会将回车符号显示成`^M`。同一个脚本执行两种相反的操作,取决于是否调用 `unix2dos` 还是(默认情况下以任何其他名称调用)`dos2unix`。

```
$ cat -v hello.txt
line one^M
line two^M
line three^M
$ ./dos2unix hello.txt
$ cat -v hello.txt
line one
line two
line three
$ ./unix2dos hello.txt
```

```
$ cat -v hello.txt
line one^M
line two^M
line three^M
$
```

也就是说，单个文件可以嵌入与其目的有关的很多隐藏信息。无论是要挂载或卸载文件系统，还是在 Unix 格式与 DOS 格式之间转换文本文件，细节信息都包含在一个文件中。如果实现细节(与 unix2dos/dos2unix 一样简单，sed 页面根据环境列出了 11 种不同的调用)发生了改变，那么只需要修改一个文件，而且遗漏其他修改的可能性也将减小。

14.2 date

date 命令是一个非常有用的工具。它最常见的用法是创建时间戳，特别是在发生日志事件的时候。syslog 实用程序为它记录的事件添加时间戳，所以/var/log/messages、syslog、auth.log 和其他这样的文件在一行开头都包含了这一关键元素。这对于弄清楚事件链，以及在不同日志文件甚至不同机器中捕获到的事件相互之间的关联非常有用。

14.2.1 date 的典型用法

date 命令在系统管理中最常见的两个用法大概是记录结果与状态消息，特别是在向文件记录日志与使用有意义的名称创建临时文件的时候。下面提供了这两种用法的脚本示例，并展示了一些 date 命令的高级特性能实现的更复杂功能。第一个脚本模拟了本章后面介绍的 logger 工具。第二个脚本使用 date 创建一组名称唯一但携带信息的日志文件。在这个例子中，时间戳用来跟踪 Web 服务器中的错误。在可以确认完全独立系统之间时差的情况下，系统之间可以进行比较，但常见的 NTP 源依然特别有用。



```
$ cat getuptime.sh
#!/bin/bash
LOG=/var/tmp/uptime.log
echo "`date`: Starting the $0 script." | tee -a $LOG
echo "`date`: Getting today's uptime reports." | tee -a $LOG
wget http://intranet/uptimes/index.php?get=today.csv >> $LOG 2>&1
echo "`date`: Getting this week's uptime reports." | tee -a $LOG
wget http://intranet/uptimes/index.php?get=thisweek.csv >> $LOG 2>&1
echo "`date`: Getting this month's uptime reports." | tee -a $LOG
wget http://intranet/uptimes/index.php?get=thismonth.csv >> $LOG 2>&1
echo "`date`: Getting this year's uptime reports." | tee -a $LOG
wget http://intranet/uptimes/index.php?get=thisyear.csv >> $LOG 2>&1
echo "`date`: Finished the $0 script." | tee -a $LOG

$ ./getuptime.sh
Tue Mar 22 14:15:01 GMT 2011: Starting the ./getuptime.sh script.
Tue Mar 22 14:15:01 GMT 2011: Getting today's uptime reports.
Tue Mar 22 14:15:07 GMT 2011: Getting this week's uptime reports.
```

```
Tue Mar 22 14:15:13 GMT 2011: Getting this month's uptime reports.
Tue Mar 22 14:15:16 GMT 2011: Getting this year's uptime reports.
Tue Mar 22 14:15:22 GMT 2011: Finished the ./getuptime.sh script.
$ cat /var/tmp/uptime.log
Tue Mar 22 14:15:01 GMT 2011: Starting the ./getuptime.sh script.
Tue Mar 22 14:15:01 GMT 2011: Getting today's uptime reports.
--2011-03-22 14:15:01-- http://intranet/uptimes/index.php?get=today.csv
Resolving intranet... 192.168.0.210
Connecting to intranet|192.168.0.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4398 (4.3K) [text/html]
Saving to: `intranet/uptimes/index.php?get=today.csv'
```

```
OK .... 100% 168M=0s
```

```
2011-03-22 14:15:07 (168 MB/s) - `intranet/uptimes/index.php?get=today.csv' saved [
4398/4398]
```

```
Tue Mar 22 14:15:07 GMT 2011: Getting this week's uptime reports.
--2011-03-22 14:15:07-- http://intranet/uptimes/index.php?get=thisweek.csv
Resolving intranet... 192.168.0.210
Connecting to intranet|192.168.0.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4398 (4.3K) [text/html]
Saving to: `intranet/uptimes/index.php?get=thisweek.csv'
```

```
OK .... 100% 173M=0s
```

```
2011-03-22 14:15:13 (173 MB/s) - `intranet/uptimes/index.php?get=thisweek.csv' save
d [4398/4398]
```

“本月”报告因为一些原因导致服务器于 14:15:13 发生错误。

```
Tue Mar 22 14:15:13 GMT 2011: Getting this month's uptime reports.
--2011-03-22 14:15:13-- http://intranet/uptimes/index.php?get=thismonth.csv
Resolving intranet... 192.168.0.210
Connecting to intranet|192.168.0.210|:80... connected.
HTTP request sent, awaiting response... 501 Internal Server Error
2011-03-22 14:15:16 ERROR 501: Internal Server Error.
```

```
Tue Mar 22 14:15:16 GMT 2011: Getting this year's uptime reports.
--2011-03-22 14:15:16-- http://intranet/uptimes/index.php?get=thisyear.csv
Resolving intranet... 192.168.0.210
Connecting to intranet|192.168.0.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4398 (4.3K) [text/html]
Saving to: `intranet/uptimes/index.php?get=thisyear.csv'
```

```
OK .... 100% 185M=0s
```

```
2011-03-22 14:15:22 (185 MB/s) - `intranet/uptimes/index.php?get=thisyear.csv' save
```

d [4398/4398]

Tue Mar 22 14:15:22 GMT 2011: Finished the ./getuptime.sh script.

\$ **tail -4 /var/log/apache2/access.log**

192.168.5.103 -- [22/Mar/2011:14:15:01 +0000] "GET /uptimes/index.php?get=today.csv HTTP/1.0" 200 4649 "-" "Wget/1.12 (linux-gnu)"

192.168.5.103 -- [22/Mar/2011:14:15:07 +0000] "GET /uptimes/index.php?get=thisweek.csv HTTP/1.0" 200 4649 "-" "Wget/1.12 (linux-gnu)"

➔ 192.168.5.103 -- [22/Mar/2011:14:15:13 +0000] "GET /uptimes/index.php?get=thismonth.csv HTTP/1.0" **501** 229 "-" "Wget/1.12 (linux-gnu)"

192.168.5.103 -- [22/Mar/2011:14:15:16 +0000] "GET /uptimes/index.php?get=this year.csv HTTP/1.0" 200 4649 "-" "Wget/1.12 (linux-gnu)"

\$

服务器访问日志中的这一行于 14:15:13 显示 501 错误。

getuptime.sh

第二个脚本对系统内存定期进行快照。当然有更强大的系统监视工具，但有时全部所需的仅仅是一个简单的 shell 脚本。这个非常简短的脚本每隔一分钟获取一次/proc/meminfo 的副本，以此查看服务器的内存使用情况。每个文件都名为 mem.(year)(month)(day).(hour)(minutes)。



可从
wrox.com
下载源代码

\$ **cat monitor.sh**

#!/bin/bash

while :

do

cat /proc/meminfo > /var/tmp/mem.`date +%Y%m%d.%H%M`

sleep 60

done

\$

monitor.sh

上面的脚本每分钟创建一个新文件。这些数据很容易经过分析后让电子表格程序读取，并显示成图表。

/var/tmp# **grep MemFree mem.***

```
mem.20110323.1715:MemFree:      131217092 kB
mem.20110323.1716:MemFree:      129300240 kB
mem.20110323.1717:MemFree:      124681904 kB
mem.20110323.1718:MemFree:      117881144 kB
mem.20110323.1719:MemFree:      120531736 kB
mem.20110323.1720:MemFree:      112337316 kB
mem.20110323.1721:MemFree:      110234640 kB
mem.20110323.1722:MemFree:      106036032 kB
mem.20110323.1723:MemFree:      91977924 kB
mem.20110323.1724:MemFree:      78725428 kB
mem.20110323.1725:MemFree:      78719628 kB
mem.20110323.1726:MemFree:      78720700 kB
mem.20110323.1727:MemFree:      78720376 kB
mem.20110323.1728:MemFree:      77418280 kB
```



```

mem.20110323.1729:MemFree:      73464744 kB
mem.20110323.1730:MemFree:      80239176 kB
mem.20110323.1731:MemFree:      78712968 kB
mem.20110323.1732:MemFree:      78717092 kB
mem.20110323.1733:MemFree:      66421840 kB
mem.20110323.1734:MemFree:      48610120 kB
mem.20110323.1735:MemFree:      36769560 kB
mem.20110323.1736:MemFree:      36693548 kB
mem.20110323.1737:MemFree:      36694500 kB
mem.20110323.1738:MemFree:      36694984 kB
mem.20110323.1739:MemFree:      36698632 kB
mem.20110323.1740:MemFree:      36703592 kB
mem.20110323.1741:MemFree:      36668052 kB
mem.20110323.1742:MemFree:      36681928 kB
mem.20110323.1743:MemFree:      36685028 kB
mem.20110323.1744:MemFree:      36686388 kB
mem.20110323.1745:MemFree:      36676812 kB
mem.20110323.1746:MemFree:      36684908 kB
mem.20110323.1747:MemFree:      36685308 kB
mem.20110323.1748:MemFree:      36685056 kB
mem.20110323.1749:MemFree:      36685088 kB
mem.20110323.1750:MemFree:      36685432 kB
mem.20110323.1751:MemFree:      36684512 kB
mem.20110323.1752:MemFree:      36684736 kB
mem.20110323.1753:MemFree:      36684564 kB
mem.20110323.1754:MemFree:      35108364 kB

```

下面这个简短的 `while` 循环使用除法将单位从 KB 扩大到 MB，再到 GB，并将含有时间的字段剪切出来，显示时使用逗号将时间与内存大小分开。最后为了显示清晰，我们将结果用管道传递给 `pr -T -4`。实际上，可以不使用 `done | pr -T -4`，而使用 `done > memory.csv`。随后，电子表格软件可以用来将它转换为表格，显示随着时间的流逝可用内存也被慢慢消耗。图 14-1 显示了内存随时间的使用情况。

```

# grep MemFree *|cut -d. -f3-|cut -d: -f1,3| while read time mem kb
> do
>   echo "${time:0:4},`expr $mem / 1024 / 1024`"
> done | pr -T -4
1715,125      1725,75      1735,35      1745,34
1716,123      1726,75      1736,34      1746,34
1717,118      1727,75      1737,34      1747,34
1718,112      1728,73      1738,34      1748,34
1719,114      1729,70      1739,34      1749,34
1720,107      1730,76      1740,35      1750,34
1721,105      1731,75      1741,34      1751,34
1722,101      1732,75      1742,34      1752,34
1723,87       1733,63      1743,34      1753,34
1724,75       1734,46      1744,34      1754,33
#

```

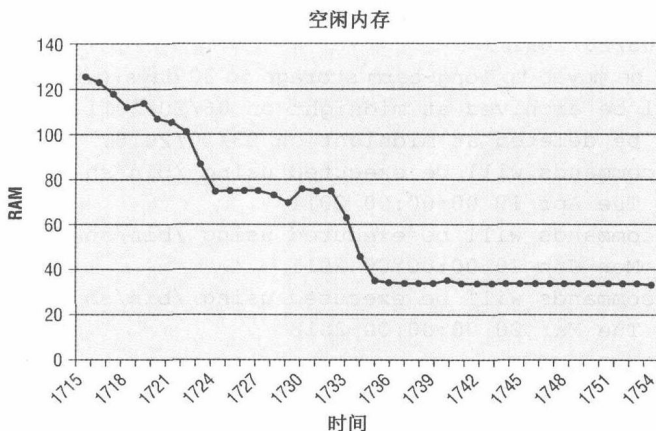


图 14-1

14.2.2 date 的一些更有趣的用法

`date` 实际上是一个非常灵活的命令，可以实现非常复杂的时间转换操作，就如同所有处理日期的 GNU 实用程序(`touch`、`at`、`batch` 等)那样。关于一些更复杂的可用功能的解释，请访问 http://www.gnu.org/software/tar/manual/html_chapter/Date-input-formats.html。下面的脚本处理对敏感数据的近线存储、归档以及最终删除这 3 种操作，方法是分别计算 30 天、3 个月以及 7 年之后的日期。如果不使用 `date` 确实会非常复杂。如果时间是 10 月、11 月或 12 月，增加 3 个月会改变年份。如果是在 1 月 29 号之后，增加 30 天可能会也可能不会到 3 月份，具体取决于是否为闰年。`date` 可以在后台为我们处理好所有这些情况。



```
$ date
Sun Mar 20 15:23:19 EDT 2011
$ cat save_records.sh
#!/bin/bash
DATEFORMAT="%m/%d/%Y"

TODAY=`date +%{DATEFORMAT}`
echo "Today is $TODAY"

# Get the three dates
LONGTERM=`date -d "30 days" "+${DATEFORMAT}"`
ARCHIVAL=`date -d "3 months" "+${DATEFORMAT}"`
DELETION=`date -d "7 years" "+${DATEFORMAT}"`

echo "Files will be moved to long-term storage in 30 days (midnight at $LONGTERM)."
echo "Files will be archived at midnight on $ARCHIVAL."
echo "They will be deleted at midnight on $DELETION."

at -f /usr/local/bin/longterm_records "$1" midnight $LONGTERM
at -f /usr/local/bin/archive_records "$1" midnight $ARCHIVAL
at -f /usr/local/bin/delete_records "$1" midnight $DELETION
```

```
$ ./save_records.sh /var/spool/data/todays_records/
Today is 03/20/2011
Files will be moved to long-term storage in 30 days (midnight at 04/19/2011).
Files will be archived at midnight on 06/20/2011.
They will be deleted at midnight on 03/20/2018.
warning: commands will be executed using /bin/sh
job 28 at Tue Apr 19 00:00:00 2011
warning: commands will be executed using /bin/sh
job 29 at Mon Jun 20 00:00:00 2011
warning: commands will be executed using /bin/sh
job 30 at Tue Mar 20 00:00:00 2018
$
```

save_records.sh

14.3 dirname

`dirname` 与 `basename` 属于同一类，但作用相反，前者从路径中返回的是目录名。当脚本不知道文件将要保存的具体位置时，`dirname` 非常有用。例如，当脚本是可以解压至用户主目录、`/tmp` 或其他任意位置的压缩包的一部分时。下面的脚本使用 `dirname $0` 非常宽泛地针对一个相当复杂的目录结构产生自身的相对路径。它使用 `etc/install.cfg` (相对于脚本的位置)来决定是否交互式地询问安装者是否接收许可证条款。



可从
wrox.com
下载源代码

```
$ cat install.sh
#!/bin/bash
ACCEPT_LICENSE=0 # may be overridden by the config file

echo "Reading configuration..."
CFG=`dirname $0`/etc/install.cfg
. $CFG
echo "Done."

mkdir `dirname $0`/logs 2>/dev/null || exit 1

if [ "$ACCEPT_LICENSE" -ne "1" ]; then
    ${PAGER:-more} `dirname $0`/LICENSE.TXT
    read -p "Do you accept the license terms?"
    case $REPLY in
        y*|Y*) continue ;;
        *) echo "You must accept the terms to install the software."
           exit 1 ;;
    esac
fi

rm -f `dirname $0`/logs/status
case `uname` in
Linux)
    for rpm in `dirname $0`/rpms/*.rpm
```

```

do
    rpm -Uvh $rpm 2>&1 | tee `dirname $0`/logs/`basename ${rpm}`.log
    echo "${PIPESTATUS[0]} $rpm" >> `dirname $0`/logs/status
done
;;
SunOS)
    for pkg in `dirname $0`/pkgs/*.pkg
    do
        pkgadd -d $pkg 2>&1 | tee `dirname $0`/logs/`basename ${pkg}`.log
        echo "${PIPESTATUS[0]} $pkg" >> `dirname $0`/logs/status
    done
    ;;
    *) echo "Unsupported OS. Only RPM and PKG formats available."
        exit 2 ;;
esac

echo
# Check for errors... grep -v "^0 " returns 1 if there *are* any lines
# which start with something other than "0 ".
grep -v "^0 " `dirname $0`/logs/status
if [ "$?" -ne "1" ]; then
    echo "Errors were encountered during installation of the above packages."
    echo "Please investigate before using the software."
else
    echo "Software installed successfully."
fi

```

install.sh

我们运行了上面的 `install.sh` 脚本，`root` 用户知道安装程序已经下载到用户 `steve` 的主目录中，而该目录不是创建 `install.sh` 的脚本编写者所能得知的。如果 `root` 变成了 `funky-1.40/` 目录，那么配置文件正确的相对路径是 `etc/install.cfg`。因为 `root` 实际上从 `/home/steve` 中调用安装程序，所以正确的相对路径应该是 `funky-1.40/etc/install.cfg`。通过使用 `dirname`，我们不必担心这些问题，当然也不必将绝对路径 `/home/steve/funky-1.40/etc/install.cfg` 硬编码到脚本中去。



可从
wrox.com
下载源代码

```

# cd ~steve
# pwd
/home/steve
# tar xzvf /tmp/funky-1.40.tar.gz
funky-1.40/
funky-1.40/install.sh
funky-1.40/rpms/
funky-1.40/rpms/fnkygroovyapps-1.40-1-x86_64.rpm
funky-1.40/rpms/fnkygroovycfg-1.40-1-x86_64.rpm
funky-1.40/rpms/fnkygroovy-1.40-1-x86_64.rpm
funky-1.40/pkgs/
funky-1.40/pkgs/FNKYgroovycfg.pkg

```

```

funky-1.40/pkgsg/FNKYgroovyapps.pkg
funky-1.40/pkgsg/FNKYgroovy.pkg
funky-1.40/etc/
funky-1.40/etc/install.cfg
funky-1.40/LICENSE.TXT
# cat funky-1.40/etc/install.cfg
ACCEPT_LICENSE=0
# echo ACCEPT_LICENSE=1 > funky-1.40/etc/install.cfg
# funky-1.40/install.sh
Reading configuration...
Done.
Preparing... ##### [100%]
1:fnkygroovy ##### [100%]
Preparing... ##### [100%]
1:fnkygroovyapps ##### [100%]
Preparing... ##### [100%]
1:fnkygroovycfg ##### [100%]
Software installed successfully.
#

```

funky-1.40.tar.gz



dirname 会去掉最后不总是有用的/*。有时可以运行`dirname \$0`; BASEDIR =`pwd`来获得目录的绝对路径。

14.4 factor

factor 工具是产生某个数的素数因子的比较智能的(尽管对于真正的密码学工作而言,它能处理的数的大小有限制)工具。素数只能被 1 或自身整除,所以素数因子不能再进行分解。这会产生一些令人惊讶的结果,像 43 674 876 546 这么大的数也只有 5 个素数。

```

$ factor 43674876546
43674876546: 2 3 7 1451 716663
$ factor 716663
716663: 716663
$

```

下面的脚本将 **factor** 的输出重新格式化以显得更加智能。脚本的实现接近一种函数式而非过程式的语言。它以一个纯答案(忽略之前的冒号及其他)列表开始,每次将前两个数相乘,然后将结果与剩下的因子传递给脚本的下一个实例。这是个对未知长度的输入进行递归循环的不错方法。如果调用 **factorize** 函数时只使用一个参数,该参数必须是最后的 \$sum,也就是说所有其他因子已经被乘出来了。为了弄清楚该函数的意义,可以将 Parsing 一行解注释来看每一步执行的操作。



```
$ cat factorize.sh
#!/bin/bash

function factorize
{
    # echo "Parsing $@"
    if [ "$#" -gt "1" ];
    then
        sum=`expr $1 \* $2`
        echo "$1 x $2 = $sum"
        shift 2
        factorize $sum $@
    fi
}

# GNU says: 72: 2 2 2 3 3
# UNIX says:
#72
# 2
# 2
# 2
# 3
# 3
# So test for GNU vs non-GNU

factor --version | grep GNU > /dev/null 2>&1
if [ "$?" -eq "0" ]; then
    factorize `factor $1 | cut -d: -f2-`
else
    factorize `factor $1 | grep -v "^${1}"`
fi
```

factorize.sh

```
$ ./factorize.sh 72
2 x 2 = 4
4 x 2 = 8
8 x 3 = 24
24 x 3 = 72
$ ./factorize.sh 913
11 x 83 = 913
$ ./factorize.sh 294952
2 x 2 = 4
4 x 2 = 8
8 x 7 = 56
56 x 23 = 1288
1288 x 229 = 294952
$ ./factorize.sh 43674876546
2 x 3 = 6
6 x 7 = 42
```

```
42 x 1451 = 60942
60942 x 716663 = 43674876546
$
```



GNU 的 `factor` 程序将所有输出显示在同一行，在最终乘积与其因子之间用冒号隔开。Unix 的 `factor` 将最终乘积显示在第一行的开头，然后每个因子占一行，并使用两个空格缩进。上面的脚本在 `--version` 输出(只会在 GNU 版本中出现)中查找 GNU 字符串以判断如何进行重新格式化。对于 GNU 而言，忽略到第一个冒号之前的内容；或者对于 `factor` 的其他实现而言，剪切掉以初始值开头的那一行。

14.5 id、groups 与 getent

`id` 命令的常见用法是用于确保脚本的运行权限的正确性。它对于加强安全机制没有特别的用处，因为脚本可以被很容易地复制与编辑。但有时可以告诉用户如果不是 `root`，脚本将不会按照预期的方式运行。例如，下面这个简短的脚本显示经过配置的网络适配器与它们的当前速度。`ethtool` 命令的执行需要 `root` 权限，所以遇到不满足权限的情况，退出程序要比不能正常工作更具意义。



可从
wtox.com
下载源代码

```
$ cat nicspeed.sh
#!/bin/bash
# This script only works for the root user.
if [ `id -u` -ne 0 ]; then
    echo "Error: This script has to be run by root."
    exit 2
fi

for nic in `ls /sbin/ifconfig | grep "Link encap:Ethernet" | \
    grep "^eth" | awk '{ print $1 }'`
do
    echo -en $nic
    ethtool $nic | grep Speed:
done
$ ./nicspeed.sh
Error: This script has to be run by root.  ←—— 当运行用户权限不够时，
# ./nicspeed.sh                               脚本不再继续运行。
eth0      Speed: 1000Mb/s
eth1      Speed: 100Mb/s
eth4      Speed: 1000Mb/s  ←—— 当运行用户是 root 时，脚本正常工作。
#
```

`nicspeed.sh`

另一个总体而言与名称服务相关，但包含密码数据库的命令是 `getent`。`getent` 从特定命名服务(如 `/etc/nsswitch` 中所示)中获取键值。这些数据库中的一些可以整体输出。`getent`

`passwd` 与 `cat /etc/passwd` 等价, 但对使用的命名服务不得而知。其他数据库则不是。`getent ethers` 需要一个参数(名称或 MAC 地址都可以接受, 因为它们都是键)。



自由与开源软件的一个好处是, 如果我们想弄清楚软件的运行原理, 通常可以很容易得到并查看源代码。深入这个话题, 我们想看看 `getent group` 如何处理组名完全是数值的情况。从 <http://ftp.gnu.org/gnu/glibc/glibc-2.13.tar.gz> 中提取 `getent.c`, 然后在第 224 行找到相关代码。

```

224  gid_t arg_gid = strtoul(key[i], &ep, 10);
225
226  if (errno != EINVAL && *key[i] != '\0' && *ep ==
'\0')
227      /* Valid numeric gid. */
228      grp = getgrgid (arg_gid);
229  else
230      grp = getgrnam (key[i]);

```

上面的代码检查键作为组名是否合法。如果合法, 则将它当成组名; 否则将其当成数字。

下面的脚本使用 `groups` 与 `id` 命令各一次。对于查找的其他所有内容, 脚本分别使用 `getent group` 或 `getent passwd`。这个脚本应当是个收集组成员信息的各种组合的有用代码库。



可从
wrox.com
下载源代码

```

$ cat groups.sh
#!/bin/bash

function get_groupname
{
    [ ! -z "$1" ] && getent group $@ | cut -d: -f1
}

function get_groupid
{
    [ ! -z "$1" ] && getent group $@ | cut -d: -f3
}

function get_username
{
    [ ! -z "$1" ] && getent passwd $1 | cut -d: -f1
}

function get_userid
{
    [ ! -z "$1" ] && getent passwd $1 | cut -d: -f3
}

```



```
function get_user_group_names
{
    [ ! -z "$1" ] && groups $@ | cut -d: -f2
}

function get_user_group_ids
{
    get_user_group_names $@ | while read groups
    do
        get_groupid $groups
    done
}

function get_primary_group_id
{
    [ ! -z "$1" ] && getent passwd $1 | cut -d: -f4
}

function get_primary_group_name
{
    [ ! -z "$1" ] && get_groupname `get_primary_group_id $@`
}

function show_user
{
    [ $# -gt 0 ] && getent passwd $@ | cut -d: -f1,5
}

function show_groups
{
    for uid in $@
    do
        echo "User $uid : Primary group is `get_primary_group_name $uid`"
        printf "Additional groups: "
        for gid in `id -G $uid | cut -d" " -f2-`
        do
            printf "%s " `get_groupname $gid`
        done
        echo
    done
}

function show_group_members
{
    for sgid in `get_groupid $@`
    do
        echo
        echo "Primary members of the group `get_groupname $sgid`"
        show_user `getent passwd | cut -d: -f1,4 | grep ":${$sgid}$" | cut -d: -f1`
        echo "Secondary members of the group `get_groupname $sgid`"
    done
}
```

```

    show_user `getent group $sgid | cut -d: -f4 | tr ',' ' '`
done
}
USERNAME=${1:-$LOGNAME}
echo "User $USERNAME is in these groups: `id -Gn $USERNAME`"
show_groups $USERNAME
show_group_members `id -G $USERNAME`

$ ./groups.sh
User steve is in these groups: sysadm support
User steve : Primary group is sysadm
Additional groups: staff support

Primary members of the group sysadm:
steve:Steve Parker
bethany:Bethany Parker
Secondary members of the group sysadm:
www:Apache Web Server
dns:Bind Name Server

Primary members of the group staff:
hr:Human Resources
Secondary members of the group staff:
steve:Steve Parker
bethany:Bethany Parker
emily:Emily Parker
jackie:Jackie Parker

Primary members of the group support:
ops1:Operator Account 1
ops2:Operator Account 2
Secondary members of the group support:
steve:Steve Parker
emily:Emily Parker
$

```

groups.sh

14.6 logger

logger 是使用大多数 Unix 与 Linux 系统中都有的 **syslog** 机制的命令行工具。使用它有一些好处，其中之一是允许非特权 **shell** 脚本向超级用户拥有的日志文件执行写入操作。也就是说，实际写入的文件由系统管理员而非脚本作者决定。这提供了额外的灵活性与定制性。



可从

wrox.com
下载源代码\$ **cat checkfs.sh**

#!/bin/bash

```

logger -t checkfs -p user.info "Starting checkfs"
df | cut -c52- | grep -v "Use%" | while read usage filesystem

```

```
do
    if [ "${usage%\\%}" -gt "85" ]; then
        logger -t checkfs -s -p user.warn "Filesystem $filesystem is at $usage"
    fi
done
logger -t checkfs -p user.info "Finished checkfs"
```

checkfs.sh

```
$ df -h
Filesystem      Size Used Avail Use% Mounted on
/dev/sda5       28G  27G  395M  99% /
tmpfs           1.5G  0    1.5G   0% /lib/init/rw
udev            1.5G 248K  1.5G   1% /dev
tmpfs           1.5G  0    1.5G   0% /dev/shm
/dev/sda3       56G  55G  1.8G  97% /iso
/dev/sda6      134G 124G  3.3G  98% /home/steve
$ ./checkfs.sh
checkfs: Filesystem / is at 99%
checkfs: Filesystem /iso is at 97%
checkfs: Filesystem /home/steve is at 98%
$
```

脚本的运行导致下面的日志消息被添加到/var/log/messages 文件中。注意，尽管脚本是以可能连读取日志文件的权限都没有的非特权用户身份运行的，日志依然可以通过 syslog 工具写入文件中。

```
Mar 29 20:14:08 goldie checkfs: Starting checkfs
Mar 29 20:14:08 goldie checkfs: Filesystem / is at 99%
Mar 29 20:14:08 goldie checkfs: Filesystem /iso is at 97%
Mar 29 20:14:08 goldie checkfs: Filesystem /home/steve is at 98%
Mar 29 20:14:08 goldie checkfs: Finished checkfs
```

14.7 md5sum

MD5 是一种基于文件内容产生相当长(128 位)的校验和的算法。它可以用来检查文件是否被暗中损坏，无论文件是在磁盘上，或者更常见的情况是在像 Internet 这样不可靠的网络中传输。例如，图 14-2 显示了 GNU.org 的 FTP 服务器上的源代码、二进制文件、文档与 md5sum 文件。在下方的窗口中，md5sum --check 命令从文件中自动读取 MD5 校验和，然后将下载文件的校验和与预期值进行比较。

两个文件产生相同的校验和是有可能的。因为实际上可能存在的文件数目有无限多个，所以就算是 MD5 值相同非常罕见，但也是不可避免的。然而对于大多数情况，如果 MD5 校验和匹配，则可以放心地说这两个文件完全相同。同样地，如果 MD5 值不匹配，那么文件数据不相同。

md5sum 的一个用法是监视文件的变化。要检测文件内容是否发生变化(即使文件总的

大小没变), 可以在怀疑被修改的前后分别存储校验和。如果校验和不相同, 则文件肯定被修改过了。这对于大型文件特别有用, 因为用文件的副本来观察是否被修改是不现实的。任何文件的 MD5 校验和总是 128 位, 且与文件大小无关。因此可以定期存储文件的校验和, 而不是实际数据的副本。

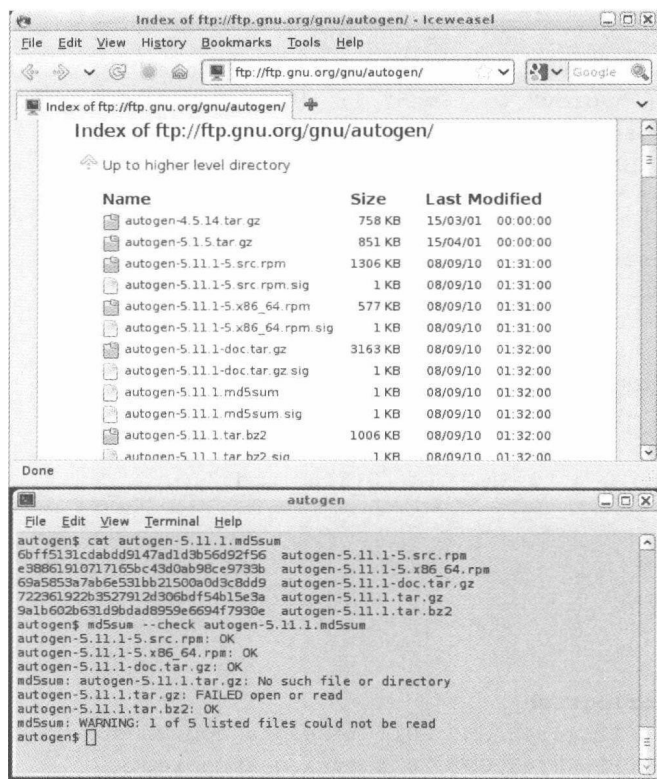


图 14-2

该脚本监视并报告文件被修改的时间。它存储正在监视的文件的 MD5 校验和, 并在发生变化时给出警告。



可从
wrox.com
下载源代码

```
# cat monitorlogs.sh
#!/bin/bash
SAVEDIR=/tmp/log.save
mkdir -p ${SAVEDIR}
cd /var/log

NOW=`date +%d%b%Y%H%M%S`
mkdir -p "$SAVEDIR" 2>/dev/null
for FILE in messages syslog dmesg daemon.log
do
    md5sum "${FILE}" | cut -d" " -f1 > "${SAVEDIR}/${FILE}.md5"
done

while :
```

```

do
NOW=`date +%d%b%Y%H%M%S`
for FILE in messages syslog dmesg daemon.log
do
prev=`cat "${SAVEDIR}/${FILE}.md5" || echo 0`
if [ -s "${FILE}" ]; then
# it exists and has content
md5=`md5sum ${FILE} | cut -d" " -f1 | tee "${SAVEDIR}/${FILE}.md5"`
if [ "$prev" != "$md5" ]; then
case "$prev" in
0) echo "`date`: $FILE appeared." ;;
*) echo "`date`: $FILE changed."
;;
esac
cp "${FILE}" "${SAVEDIR}/${FILE}.$NOW"
fi
else
# it doesn't exist; did it exist before?
if [ "$prev" != "0" ]; then
echo "`date`: $FILE disappeared."
echo 0 > "${SAVEDIR}/${FILE}.md5"
fi
fi
done
sleep 30
done

# ./monitorlogs.sh
Fri Feb 11 11:44:45 GMT 2011 messages appeared.
Fri Feb 11 11:44:45 GMT 2011 syslog appeared.
Fri Feb 11 11:44:45 GMT 2011 dmesg appeared.
Fri Feb 11 11:44:45 GMT 2011 daemon.log appeared.
Fri Feb 11 11:45:15 GMT 2011 messages changed.
Fri Feb 11 11:45:15 GMT 2011 syslog changed.
Fri Feb 11 11:46:15 GMT 2011 messages changed.
Fri Feb 11 11:46:15 GMT 2011 syslog changed.
^C
#

```

monitorlogs.sh

14.8 mkfifo

先入先出(First-In First-Out, FIFO)文件是命名管道。像用来连接独立命令的输入与输出的一般管道一样, FIFO 管理进程间的输入与输出。然而, 任何进程都可以(如果文件系统权限允许)从该管道读取或向管道写入。这允许多进程之间进行相互通信, 甚至不知道哪个进程会获取它们发送的数据, 或者接收哪个进程发送的数据。`mkfifo` 命令建立一个 FIFO, 并同时可选地进行权限设置。

14.8.1 主与从

这种进程间通信以易于使用与理解的方式提供了非常有用的多任务功能。主(master)脚本偶尔会发出命令。它的很多从(minion)脚本之一选中一些命令并运行。这一工作可能要花费一些时间才能完成(这些客户端有很多 `sleep` 语句来确保要花费一些时间),但主脚本可以向队列分配任务,然后返回到循环中。然而,当所有从脚本(或客户端)都运行起来时,主脚本向 FIFO 的回显要等到有一个客户端响应时才会返回,所以说主脚本当时是被阻塞了。如图 14-3 所示,当第一个客户端(中间左边的那个)选择了 `quit` 命令,该客户端便退出了。当剩下的客户端响应第二个 `quit` 命令退出后,不再有进程侦听这个管道,所以主脚本要等到某个客户端执行它的上一个命令时才会显示出命令行提示符。

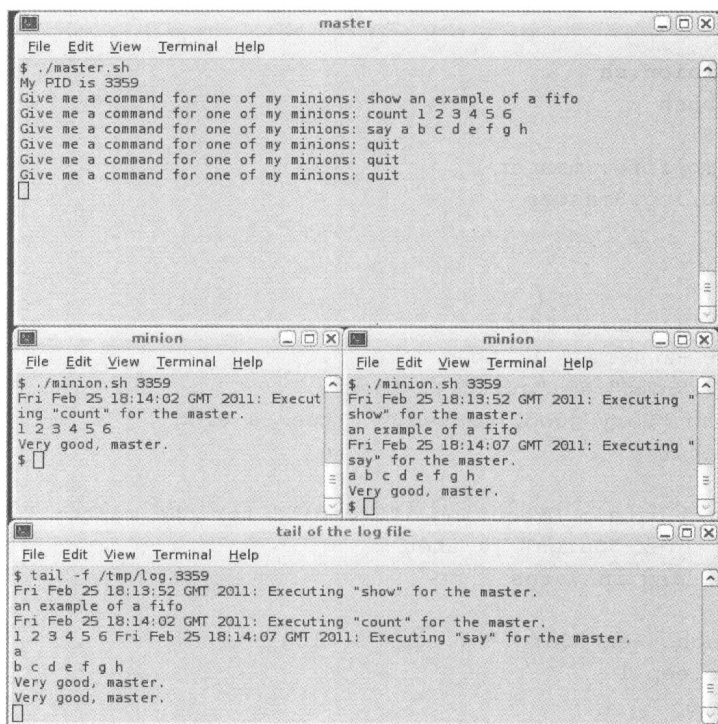


图 14-3

当然,这些“命令”是一种编造的语言。从脚本只会将它们读取的第一个单词当成命令,然后回显剩下的单词,每次一个,间隔为 1 秒。



可从
wrox.com
下载源代码

```
$ cat master.sh
#!/bin/bash
pid=$$
fifo=/tmp/fifo.$pid
log=/tmp/log.$pid
> $log
```

```

echo "My PID is $pid"
mkfifo $fifo

while :
do
    echo -en "Give me a command for one of my minions: "
    read cmd
    echo $cmd > $fifo
done

rm -f $log $fifo

```

master.sh

```

$ cat minion.sh
#!/bin/bash
master=$1
fifo=/tmp/fifo.$master
log=/tmp/log.$master

while :
do
    read cmd args < $fifo
    if [ ! -z "$cmd" ]; then
        if [ "$cmd" == "quit" ]; then
            echo "Very good, master." | tee -a $log
            exit 0
        fi
        echo "`date`: Executing \"${cmd}\" for the master." | tee -a $log
        if [ ! -z "$args" ]; then
            for arg in $args
            do
                echo -en "$arg " | tee -a $log
                sleep 1
            done
            echo | tee -a $log
        fi
        sleep 10
    done
done
$

```

minion.sh

运行时，从脚本显示到输出，也(通过不常用但非常方便的 `tee -a` 实用程序)记录到日志文件。底部的窗口用来观察写入的内容。注意在图 14-3 中，命令 `say a b c d e f g h` 就在 `count 1 2 3 4 5 6` 结束之前开始运行。`count` 命令中最后的 `echo` 记录到时间戳之后，并且由 `say` 命令写入一个初始的 `a`。这些脚本的运行示例如图 14-3 所示。



没有任何技术上的理由让主、从脚本使用主脚本的 PID 命名它们使用的文件的名称。文件可以就是简单的/tmp/master.fifo 或者任意其他名称。该文件可能是文件系统中处于特定位置并经过/opt/master/comms/master-minion 这样的应用程序文档化的永远不变的部分。使用/tmp 与 PID 作为标签通常可以很方便地确保两个并发运行的主-从 FIFO 可以同时存在而不相互影响。

14.8.2 颠倒顺序

反过来说,可能有很多独立的进程向 FIFO 进行写操作,由一个(或多个)主进程从管道的另一头读取。一组进程可以用来收集要完成的任务片段,另一些可以像前面例子中的从进程一样选择下一个待完成任务。下面的脚本有多个收集进程,它们并发地从不同文件系统中读取,只要找到 zip 文档就写入 FIFO。在这种场景下,主进程是从 FIFO 读取的脚本。下面的脚本从 FIFO 中读取 zip 文件,并查找包含单词 chapter 的文件名。



可从
wrox.com
下载源代码

```
#!/bin/bash
fifo=/tmp/zips.fifo
rm $fifo
mkfifo $fifo
searchstring=$@

while read filename
do
    unzip -l "${filename}" | grep $searchstring > /dev/null 2>&1
    if [ "$?" -eq "0" ]; then
        echo "Found \"$searchstring\" in $filename"
    fi
done < $fifo
echo "Finished."
```

zip-master.sh

zip-gatherer.sh 脚本包含一个 searchfs 函数。这个函数在给定的文件系统(所以搜索/不会与独立的/home 文件系统重叠)中进行查找。脚本在后台对找到的每个 crypt、ext 或者 fuseblk 类型的文件系统执行一个该函数的实例。



可从
wrox.com
下载源代码

```
#!/bin/bash
fifo=/tmp/zips.fifo

function searchfs
{
    temp=`mktemp`
    find ${1} -mount -type f -iname "*zip" -exec file {} \; \
        | grep "Zip archive data" | cut -d: -f1 > $temp
    cat $temp > $fifo
    rm -f $temp
}
```



```

    echo "Finished searching ${1}."
}

for filesystem in `mount -t crypt,ext3,ext4,fuseblk | cut -d" " -f3`
do
    echo "Spawning a child to search $filesystem"
    searchfs $filesystem &
done
# Wait for children to complete
wait
# send an EOF to the master to close the fifo
printf "%c" 04 > $fifo

```

zip-gatherer.sh

`searchfs` 函数向一个临时文件进行写操作，然后将该文件转储到 FIFO 中。这有利于一致性，因为 `find` 命令可能要运行比较长的时间，并且该函数的不同实例可能彼此重叠，导致 FIFO 中文件名混乱。

`zip-gatherer.sh` 脚本最后向 FIFO 发送一个 04(EOT)字符来结束。这样便关闭了文件，导致 `zip-master.sh` 中的 `while read` 循环退出。`zip-gatherer.sh` 使用不带参数的 `wait` 命令来确保其所有子进程完成各自的搜索。`wait` 命令要在脚本所有子进程都完成后才会返回。

`zip-gatherer.sh` 的输出如下所示。/下的一些目录对非特权用户来说是不可访问的，所以会显示 `Permission denied` 消息。通过将标准错误定向至 `/dev/null` 可以很好地将它隐藏起来。脚本主体生成后台进程对找到的每个文件系统进行搜索，然后每个进程将搜索结果写入同一个共享 FIFO 中，为主脚本的处理做准备。

```

$ ./zip-gatherer.sh
Spawning a child to search /
Spawning a child to search /windows
Spawning a child to search /home/steve
find: `/var/lib/php5': Permission denied
find: `/var/lib/polkit-1': Permission denied
find: `/var/lib/sudo': Permission denied
find: `/var/lib/gdm': Permission denied
find: `/var/cache/system-tools-backends/backup': Permission denied
find: `/var/cache/ldconfig': Permission denied
find: `/var/spool/exim4': Permission denied
find: `/var/spool/cups': Permission denied
find: `/var/spool/cron/atspool': Permission denied
find: `/var/spool/cron/atjobs': Permission denied
find: `/var/spool/cron/crontabs': Permission denied
find: `/var/run/exim4': Permission denied
find: `/var/run/cups/certs': Permission denied
find: `/var/log/exim4': Permission denied
find: `/var/log/apache2': Permission denied
find: `/home/steve/lost+found': Permission denied
find: `/etc/ssl/private': Permission denied
find: `/etc/cups/ssl': Permission denied

```

```
find: `/lost+found': Permission denied
find: `/root': Permission denied
Finished searching /.
Finished searching /home/steve.
Finished searching /windows.
$
```

zip-master.sh 脚本的输出如下面的代码片段所示。actionis.zip 文件不是合法的 zip 文件，所以 unzip 显示错误。为了获得灵活性，zip-master.sh 在其命令行中使用 grep 参数，这样我们可以指定搜索对象。也就是说，-i 参数可以传递给 grep 表示不区分大小的搜索。

```
$ ./zip-master.sh -i chapter
Found "-i chapter" in /iso/E19787-01.zip
Found "-i chapter" in /home/steve/sc32/10_x86/125509-08.zip
Found "-i chapter" in /home/steve/Part I-3feb.zip
Found "-i chapter" in /home/steve/sc33/E19680-01.zip
Found "-i chapter" in /home/steve/Part I.zip
  End-of-central-directory signature not found. Either this file is not
  a zipfile, or it constitutes one disk of a multi-part archive. In the
  latter case, the central directory and zipfile comment will be found on
  the last disk(s) of this archive.
unzip: cannot find zipfile directory in one of /home/steve/fonts/actionis.zip or
/home/steve/fonts/actionis.zip.zip, and cannot find /home/steve/fonts/actio
nis.zip.ZIP, period.
Found "-i chapter" in /home/steve/Part I-jan3.zip
Finished.
$
```

14.9 联网

联网是 Unix 与 Linux 系统的中心。有很多与联网相关的内置命令。它们全都可以编写脚本，其中有些比其他更容易。按照传统的观点，telnet 与 ping 编写脚本不太方便，所以这里给出它们的一些用法。另外，netcat 是一个更清晰、更易于编写脚本的 telnet 替代品。本节还介绍一些与不同类型 Internet 服务器对话的基本技术。最后，安全通信是如今 Internet 的关键部分，甚至可信赖网络中的内部通信也经常进行加密。加密涉及数学方面的知识，但使用起来并不是非常复杂与困难。本节还涵盖了使用 OpenSSL 工具套件的各种方法，不仅包含了 SSH 协议，还有底层的 SSL 连接本身，因为这些工具使用起来比通常想象的要容易得多。

14.9.1 telnet

telnet 是一个过时且不安全的通过网络登录远程系统的协议。然而，telnet 客户端仍然是非常有用的网络测试工具，因为它要做的只是在两个系统之间来回发送文本(尽管它将一些字符当成特殊的协议信息，但对于测试像 HTTP、SMTP、POP 与 IMAP 这样基于文本的协议已经够用)。使用一条简单的 telnet 命令可以很容易地对 Web 服务器进行测试。在下面

的例子中, `www.example.com` 的服务器给出一个 302 状态码, 表示重定向至 `http://www.iana.org/domains/example/` (HTTP 状态码的定义见 `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`), 但实际上我们可以把它当成到服务器的成功连接。400 类与 500 类的 HTTP 状态码通常表示有问题, 它们分别表示请求或服务器有问题。数字越小越好, 如 200 表示正常, 连接完全成功。



在交互式测试 HTTP 服务器时, 最容易的是指定 HTTP/1.0 而非 HTTP/1.1, 因为 1.0 版本的协议不需要额外的协议头。另外, 记得在请求之后发送一个空白行(也就是按 Return 键两次), 表示请求头部的结束。

```
$ telnet www.example.com 80
Trying 192.0.32.10...
Connected to www.example.com.
Escape character is '^]'.
GET http://www.example.com/ HTTP/1.0

HTTP/1.0 302 Found
Location: http://www.iana.org/domains/example/
Server: BigIP
Connection: close
Content-Length: 0

Connection closed by foreign host.
$
```

我们还可以使用 `telnet` 客户端根据恰当的协议直接与 SMTP、POP 和 IMAP 服务器对话, 尽管这些协议越来越多地默认就是被加密的。与加密服务通信的内容见本章后面的 14.9.5 节; OpenSSL 不像一般认为的那么难用, 而且编写脚本甚至更容易。

14.9.2 netcat

`netcat` 工具可以在网络中传输数据。它与 `telnet` 客户端非常相似, 但不会有所区别地处理特殊字符(如 EOF)。这意味着可以用它在网络上传输二进制数据, 并且不会破坏数据包。`netcat` 的用法很多, 包括用与 `telnet` 客户端一样的方式测试连接、端口扫描与文件传输。

1. 使用 netcat 测试连接

`netcat` 可以用来与基于文本协议(如 HTTP、SNMP、POP 与 IMAP)的服务器通信, 方式与 `telnet` 相似。它不会用不同的方式处理 EOF, 并且不会显示自己的关于转义字符之类的消息, 因此 `netcat` 的输出可以直接保存到文件。

```
$ telnet www.example.com 80
Trying 192.0.32.10... ◀——— 下面 3 行来自 telnet 而不是服务器。
Connected to www.example.com.
Escape character is '^]'.

```

```
HEAD / HTTP/1.0
```

```
HTTP/1.0 302 Found
Location: http://www.iana.org/domains/example/
Server: BigIP
Connection: close
Content-Length: 0
```

```
Connection closed by foreign host.
```

```
$ netcat www.example.com 80
```

```
HEAD / HTTP/1.0
```

```
HTTP/1.0 302 Found ←———— 所有输出都来自服务器。
```

```
Location: http://www.iana.org/domains/example/
Server: BigIP
Connection: close
Content-Length: 0
```

```
$
```

2. 使用 netcat 进行端口扫描

对于端口扫描，**nmap** 是比 **netcat** 功能强大得多的工具，但后者也可作为端口扫描器。另外，**netcat** 的输出非常简单，所以在自动化脚本中分析 **netcat** 输出要容易得多。

```
$ nmap 192.168.0.210
```

```
Starting Nmap 5.00 ( http://nmap.org ) at 2011-03-24 19:41 GMT
Interesting ports on intranet (192.168.0.210):
Not shown: 997 closed ports
PORT STATE SERVICE
22/tcp open  ssh
80/tcp open  http
111/tcp open  rpcbind
```

```
Nmap done: 1 IP address (1 host up) scanned in 0.08 seconds
```

```
$ netcat -vz 192.168.0.210 1-1024
```

```
intranet [192.168.0.210] 111 (sunrpc) open
intranet [192.168.0.210] 80 (www) open
intranet [192.168.0.210] 22 (ssh) open
```

```
$
```

3. 使用 netcat 传输数据

netcat 还可以用来在系统之间传输数据。因为它直接获取数据，所以可以将输出直接写到文件。在接收端，启动 **netcat -l**(侦听)并指定端口号。用管道传递给 **pv** 不是必须的，但能很好地显示进度状态。**pv -t** 显示管道的活动时间，**pv -b** 显示通过管道的字节数目。

```
recipient$ netcat -l -p 8888 | pv -t > fedora.iso.gz
0:39:17
$
```

在发送端，我们将文件 `cat` 到 `netcat` 中。再一次将 `pv` 放到管道中，这样可以方便地显示进度。另外，因为 `netcat` 忽略 EOF，所以没有比较好的方式来判断传输的结束。因此，当传输结束时，有必要(在收发两端)使用 `Ctrl-C` 来终止连接。一些发行版为 `netcat` 自动启用 `-q` 选项，但对于传输二进制文件用处不大。

```
sender$ cat /iso/Fedora-14-i686-Live-Desktop.iso.gz | pv -b | netcat recipient 8888
669MB
^C
$
```

更进一步的话，我们可以将整个压缩包或者 `cpio` 归档包发送到 `netcat`。在下面的例子中，我们使用 `tar` 的 `-v` 标志，这样随着文件写向归档包与从归档包中提取，文件会显示在收发两端。

```
recipient$ netcat -l -p 8888 | pv -t | tar xvf -
iso/:01
iso/debian-504-amd64-DVD-1.iso
iso/solaris-cluster-3_3-ga-x86.zip
iso/solaris-cluster-3_3-ga-sparc.zip
0:32:21
$
sender$ tar cvf - /iso | pv -b | netcat recipient 8888
tar: Removing leading `/' from member names
/iso/
/iso/debian-504-amd64-DVD-1.iso
/iso/solaris-cluster-3_3-ga-x86.zip
/iso/solaris-cluster-3_3-ga-sparc.zip
4.52GB
^C
$
```

14.9.3 ping

`ping` 是一个基本的网络诊断工具。它向另一个主机发送 `ICMP` 包，并请求一个回显响应。如果远程主机选择响应(除非系统根本没有反应或者是防火墙，否则应当会响应)，则会返回一个 `ICMP` 回复。这主要用于判断远程主机是否处于活动状态，但也有其他副作用。在主机处于同一子网的情况下，主要的副作用是会得到远程主机的 `MAC`(以太网)地址并添加到本地系统的 `ARP` 缓存中，可以用 `arp -a` 显示出来。传统 `ping` 实现的缺点之一是发送 4 个包后要用很长的时间等待一个响应。测试整个 C 类子网通常意味着需要下面这样一个脚本：



可从
wrox.com
下载源代码

```
$ cat simpleping.sh
#!/bin/bash
LOG=ping.log
PREFIX=192.168.1
```

```

i=1

while [ "$i" -lt "8" ]
do
    echo "Pinging ${PREFIX}.${i}"
    ping ${PREFIX}.${i} > /tmp/ping.${i} 2>&1 &
    sleep 2
    kill -9 $!
    grep "^64 bytes from" /tmp/ping.${i}
    if [ "$?" -eq "0" ]; then
        echo "${PREFIX}.${i} is alive" | tee -a $LOG
    else
        echo "${PREFIX}.${i} is dead" | tee -a $LOG
    fi
    rm -f /tmp/ping.${i}
    i=`expr $i + 1`
done

$ ./simpleping.sh
Pinging 192.168.1.1
./ping.sh: line 20:  9641 Killed      ping ${PREFIX}.${i} > /tmp/ping.${i} 2>&1
192.168.1.1 is alive
Pinging 192.168.1.2
./ping.sh: line 20:  9651 Killed      ping ${PREFIX}.${i} > /tmp/ping.${i} 2>&1
192.168.1.2 is dead
Pinging 192.168.1.3
./ping.sh: line 20:  9658 Killed      ping ${PREFIX}.${i} > /tmp/ping.${i} 2>&1
192.168.1.3 is alive
Pinging 192.168.1.4
./ping.sh: line 20:  9665 Killed      ping ${PREFIX}.${i} > /tmp/ping.${i} 2>&1
192.168.1.4 is dead
Pinging 192.168.1.5
./ping.sh: line 20:  9675 Killed      ping ${PREFIX}.${i} > /tmp/ping.${i} 2>&1
192.168.1.5 is dead
Pinging 192.168.1.6
./ping.sh: line 20:  9682 Killed      ping ${PREFIX}.${i} > /tmp/ping.${i} 2>&1
192.168.1.6 is dead
Pinging 192.168.1.7
./ping.sh: line 20:  9689 Killed      ping ${PREFIX}.${i} > /tmp/ping.${i} 2>&1
192.168.1.7 is dead
$ cat ping.log
192.168.1.1 is alive
192.168.1.2 is dead
192.168.1.3 is alive
192.168.1.4 is dead
192.168.1.5 is dead
192.168.1.6 is dead
192.168.1.7 is dead
$

```

simpleping.sh

这个脚本只测试了网络上的前 7 个设备，并且浪费了大量的显示资源。脚本最好可以指定一个超时时间与数据包的最大数量。GNU 的 ping 实现提供了这两种功能，分别使用 -w 与 -c 选项。这样一来，ping 测试变得更简单、明了与快速。上面的测试运行了大约 8 分钟。下面的则是大约 1 秒(一般会 shorter，因为单个 ping 请求通常不会花费整整一秒的时间)。要提高准确性，可以将 -w1 提高到 -w2，这样完成要花 2 秒，但给主机 2 秒而不是 1 秒的时间响应。我们还可以使用 -c2 发送两个包来防止第一个包丢失。time 命令显示命令运行使用的(实际)时间。在下面的测试运行中，整个过程只用了 1/4 秒多一些。



可从
wrox.com
下载源代码

```
$ cat ping2.sh
#!/bin/bash
LOG=ping.log
PREFIX=192.168.1

for i in `seq 1 254`
do
    ping -c1 -w1 ${PREFIX}.${i} && \
        echo "${PREFIX}.${i} is alive" | tee -a $LOG || \
        echo "${PREFIX}.${i} is dead" | tee -a $LOG &
done
$

$ time ./ping2.sh > /dev/null 2>&1

real    0m0.255s
user    0m0.008s
sys     0m0.028s
$
$ wc -l ping.log
254 ping.log
$ grep alive ping.log
192.168.1.1 is alive
192.168.1.3 is alive
192.168.1.10 is alive
$
```

ping2.sh



还有一种技术是广播 ping(ping -b)。它向本地子网的广播地址发送一个 ping 请求。子网的每个节点都应当响应该 ping 请求。这样会更简单，尽管个别节点可能经过配置后忽略广播请求，但它们还是会响应经过剪裁的 ping 请求。

14.9.4 编写 ssh 与 scp 脚本

在以前，rcp 是一个在系统之间复制文件的实用工具。我们可以很容易地进行设置，使得 shell 脚本可以从一台机器向另一台机器复制文件，而且不会被问到一些难以回答的问

题。现在的网络已不再安全，所以 rcp、rlogin、rsh 以及它们毫无安全措施的友类工具(一般称为 r 类工具)都被废弃了。



尽管 rsync 工具的名称以表示 remote 的 r 开头,但该工具不属于 r 类工具,并可以在经过配置后使用 ssh 来进行安全认证与加密。

很多人都对交互式使用 scp 比较熟悉,但它也可以用来提供安全的无口令认证。其关键是公共密钥基础设施(Public Key Infrastructure, PKI)。PKI 表示我们可以拥有非对称密钥,即保密的那个被称为私钥。另一个密钥可以安全地在任何人之间共享,甚至是不受信任的对手。这个密钥称为公钥。使用公钥加密的数据可以用私钥进行解密,反之亦然。这实际上意味着,我们不用将密钥透漏给任何人就可以向远程主机证明我们是私钥的所有者。这是很多其他特性中无口令登录的基础,包括安全的 Web 浏览。



对私钥使用空白口令会拒绝 PKI 的整个认证。有时这样做没有问题——如果所有需要的只是加密,那么认证就无关紧要——但对于认证目的,为私钥提供一个好的口令是很有必要的。

ssh-keygen 工具创建一对密钥。所有随后要做的是将公钥(id_rsa.pub)复制(或追加)到远程服务器的 ~/.ssh/authorized_keys 中,并将权限设置正确(~/.ssh 目录为 0700, ~/.ssh/authorized_keys 为 0600)。ssh-copy-id 工具可以为我们完成所有这些任务。

```
home$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/steve/.ssh/id_rsa): <enter>
Created directory '/home/steve/.ssh'.
Enter passphrase (empty for no passphrase): ssh-password
Enter same passphrase again: ssh-password
Your identification has been saved in /home/steve/.ssh/id_rsa.
Your public key has been saved in /home/steve/.ssh/id_rsa.pub.
The key fingerprint
is:
28:17:fd:fe:df:60:8a:fa:9e:17:c0:94:8c:5f:e2:35
home$
home$ ssh-copy-id -i ~/.ssh/id_rsa.pub example.com
steve@example.com's password: example.com-password
Now try logging into the machine, with "ssh 'example.com' ", and check in:

.ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.
home$ ssh example.com
Enter passphrase for key '/home/steve/.ssh/id_rsa': ssh-password
```



```
Last login: Fri Dec 11 11:01:33 2009 from home
example.com$ ls -l .ssh/authorized_keys
-rw----- 1 steve user 395 Jun 3 2011 authorized_keys
example.com$
```

下一步是将密钥添加到环境中。如果正在运行图形化会话(如 GNOME 或 KDE), 那么 SSH 代理应当已经运行。如果不是图形化会话, 则需要让当前 shell 分析 ssh-agent 命令的输出。它会给出与 ssh-agent 对话所需的所有设置, 调用结束后会自动在后台启动。然后, 我们可以手动地使用 ssh-add 命令向代理添加密钥。添加时会提示输入密钥口令, 然后将密钥存储在内存中用于将来的连接。

```
home$ eval `ssh-agent`
Agent pid 3996
home$ ps -fp 3996
UID      PID PPID C STIME TTY      TIME CMD
Steve    3996  1 0 19:43 ?        00:00:00 ssh-agent
home$ ssh-add
Enter passphrase for /home/steve/.ssh/id_rsa:
Identity added: /home/steve/.ssh/id_rsa (/home/steve/.ssh/id_rsa)
home$ ssh steve@example.com
steve@example.com:~$ uname -n
example.com
steve@example.com:~$
```

整个设置只用几分钟时间, 并且能为远程连接提供安全的基础设施。这可以用来在机器之间提供类似 rcp 的文件无缝复制。下面的脚本检查认证机制正常工作, 如果不正常则显示错误消息。一旦确认基础设施能正常工作, 脚本使用 scp 将命令行中列出的文件复制到远程主机。因为调用 shell 有一个活动的 ssh-agent, 所以在执行过程中没有询问口令。



可从
wrox.com
下载源代码

```
$ cat scp.sh
#!/bin/bash
user=$1
host=$2
shift 2
files=$@

echo "Testing connection to ${host}..."
ssh -n -o NumberOfPasswordPrompts=0 ${user}@${host}
if [ "$?" -ne "0" ]; then
    echo "FATAL: You do not have passwordless ssh working."
    echo "Try running ssh-add."
    exit 1
fi

echo "Okay. Starting the scp."
scp -B ${files} ${user}@${host}:
if [ "$?" -ne "0" ]; then
    echo "An error occurred."
```

```

else
    echo "Successfully copied $files to $host"
fi

echo "I can do ssh as well."
ssh ${user}@${host} ls -l ${files}

$ ./scp.sh wronguser example.com hosts scp.sh data
Testing connection to example.com...
Pseudo-terminal will not be allocated because stdin is not a terminal.
Permission denied (publickey,password,keyboard-interactive).
FATAL: You do not have passwordless ssh working.
Try running ssh-add.
$ ./scp.sh steve example.com hosts scp.sh data
Testing connection to example.com...
Pseudo-terminal will not be allocated because stdin is not a terminal.
Okay. Starting the scp.
Hosts                                100% 479          0.5KB/s      00:00
scp.sh                               100% 436          0.4KB/s      00:00
data                                 100% 4096KB 105.0KB/s  00:39
Successfully copied hosts scp.sh data to example.com
I can do ssh as well.
-rw-r--r-- 1 steve user      479 Mar 21 14:51 hosts
-rw-r--r-- 1 steve user 4194304 Mar 21 14:52 data
-rwxr-xr-x 1 steve user      501 Mar 21 14:51 scp.sh
home$

```

scp.sh

就是这么简单。借助 `ssh`，我们可以以一种安全的方式设置自动化复制，甚至通过脚本在远程机器上运行命令。`ssh` 是一个非常有用的工具，而做到安全所需的全部就是对口令进行保密，并且不让任何人接手自己的登录会话。

14.9.5 OpenSSL

OpenSSL 是管理安全套接字层(Secure Sockets Layer, SSL)连接的库。它提供两个关键的优势：认证与加密。本节主要介绍加密方面的内容。对于认证，密钥是由认证机构(Certificate Authority, CA)签名的。认证机构由客户端软件(一般是 Web 浏览器或邮件客户端)识别。余下的部分无论是由 CA 或者用户个人签名(自签名证书)都一样。添加 SSL 会增加额外的复杂度。例如，不可能使用 `telnet` 建立 SSL 连接。SSL 连接的建立有一定的复杂度，但二进制文件 `openssl` 对 `s_client` 命令进行了有效封装。它在后台实现了所有 SSL 协议，并尽可能简单地提供安全传输机制。

`openssl s_client` 工具在后台完成所有的 SSL 握手并显示结果。在下面的代码片段中，对 `www.google.com` 端口 443 的连接交换了证书，并建立了一个安全连接。这样一来，HTTP 会话与之前显示的无加密会话一模一样。302 状态码用于重定向至 `google.com` 上的另一页面。然而，窃听者现在看见的将是经过加密的数据。



同样的技术可以用于连接到安全的 SMTP、IMAP、POP 以及其他基于文本的服务。

```
$ openssl s_client -connect www.google.com:443
CONNECTED(00000003)
depth=1 /C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
verify error:num=20:unable to get local issuer certificate
verify return:0
---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
  i:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
 1 s:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
  i:/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDITCCAoqgAwIBAgIQI9+89q6RUm0PmqPfQDQ+mjANBgkqhkiG9w0BAQUFADBM
MQswCQYDVQQGEWJaQTElMCMGA1UEChMcVGhhd3RlIENvbnN1bHRpbmcgKFB0eSkq
THRkLjEWMBQGA1UEAxMNVGhhd3RlIFNHQyBDQTAEFw0wOTEyMTgwMDAwMDBaFw0x
MTEyMTgyMzU5NTlaMGgxGzAJBgNVBAYTA1VTMRMwEQYDVQQIEwpDYWxpZm9ybmlh
MRYwFAYDVQQHFA1Nb3VudGFpbWV3MRMwEQYDVQQKFAPhb29nbGUgSW5jMRcw
FQYDVQQDFA53d3cuZ29vZ2x1LmNvbTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkC
gYEA6PmGD5D6htffvXImttDEAoN4c9kCKO+IRTn7EOh8rqk41XXGOOsKFQebg+jN
gtXj9xVoRaELGYW84u+E593y17iYwqG7tcFR39SDAqc9BkJb4SLD3muFXxZW2k6L
05vuuWciKh0R73mkszeK9P4Y/bz5RiNQ1/Os/CRGK1w7t0UCAwEAAaOB5ZCB5DAM
BgNVHRMBAf8EAjAAMDYGA1UdHwQvMC0wK6ApoCeGJWh0dHA6Ly9jcmwudGhhd3Rl
LmNvbS9UaGF3dGVTR0NDQS5jcmwwKAYDVR0lBCEwHwYIKwYBBQUHAwEGCCsGAQUF
BwMCBgIghkgBhvCBBAEwcgYIKwYBBQUHAQEZjBkMCIGCCsGAQUFBzABhhZodHRw
Oi8vb2Nzc50aGF3dGUuY29tMD4GCCsGAQUFBzACHjJodHRwOi8vd3d3LnRoYXN0
ZS5jb20vcmwvbm3NpdG9yeS9UaGF3dGVfU0dDX0NBmNydDANBgkqhkiG9w0BAQUF
AAOBGQCfQ89bxFapsb/isJr/aiEdLRLDLE5a+RLizrmCUI3nHX4adpaQedEkUjh5
u2ONgJd8IyAPkU0Wueru9G2Jysa9zCRolknBzipYvzwY4OA8Ys+WAi0oR1A04Se6
z5nRUP8pJcA2NhUzUnC+MY+f6H/nEQyNv4SgQhqAibAxWEEHXw==
-----END CERTIFICATE-----
subject=/C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
issuer=/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
---
No client certificate CA names sent
---
SSL handshake has read 1772 bytes and written 307 bytes
---
New, TLSv1/SSLv3, Cipher is RC4-SHA
Server public key is 1024 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
```

```

Protocol : TLSv1
Cipher   : RC4-SHA
Session-ID : 2E0A49E8A432E2EE45B1449744BFF2017F6AC0F7F2CB477F122770666D0FD5A7
Session-ID-ctx:
Master-Key : 1D0E53FDF5D5B4C50FD7040855DEAD1F59F4A31FADDEFA95D33B53FB066FE54A1055
BD40C472CEF54BD0F67155C6609C2
Key-Arg   : None
Start Time : 1298235973
Timeout    : 300 (sec)
Verify return code: 20 (unable to get local issuer certificate)
---
GET https://www.google.com/ HTTP/1.0 ←———— SSL 握手之后就与之前显示的
telnet 会话一样了。

HTTP/1.0 302 Found
Location: https://encrypted.google.com/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie: PREF=ID=6d0ef1135c2f2888:FF=0:TM=1298235982:LM=1298235982:S=CukIbpibhIa
tYpgM; expires=Tue, 19-Feb-2013 21:06:22 GMT; path=/; domain=.google.com
Set-Cookie: NID=44=Rg2UzMltwCSAtFrZwCB6niEX7vQjKa25eR3qkKaEtqP6Nx5Lb01PM9Rk1lUgZ5u
XZ3sg4kEmp7lpoP2U8knxgZHPBM7Tz7kbD087T9iHSHpThgdtcMXeKIb7kItvnqO; expires=Mon, 22-A
ug-2011 21:06:22 GMT; path=/; domain=.google.com; HttpOnly
Date: Sun, 20 Feb 2011 21:06:22 GMT
Server: gws
Content-Length: 226
X-XSS-Protection: 1; mode=block

<HTML><HEAD><meta http-equiv="content-type" content="text/html; charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="https://encrypted.google.com/">here</A>.
</BODY></HTML>
read:errno=0
$

```

二进制文件 `openssl` 还能够运行一个非常基本的安全的 Web 服务器。它传递与程序运行所在的本地目录相关的文件，所以当在 `/var/tmp` 中运行时，对 `/README` 的请求将会返回文件 `/var/tmp/README`。这并不适合于在生产环境中使用，但对于测试 SSL 客户端的连接非常有用。这里的内容不是给 SSL 初学者讲的，但下面的代码会为我们生成一个自签名的证书(密码是 `welcome123`)，用于 `OpenSSL` 服务器。



我们可以在创建 `server.pem` 之后扔掉 `server.key` 与 `server.crt` 文件。只有 `server.pem` 是必需的。

```

$ openssl genrsa -des3 1024 > server.key
Generating RSA private key, 1024 bit long modulus

```

```

.....+++++
.....+++++
e is 65537 (0x10001)
Enter pass phrase: welcome123
Verifying - Enter pass phrase: welcome123
$ openssl req -new -key server.key -x509 -days 3650 -out server.crt
Enter pass phrase for server.key: welcome123
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:New York
Locality Name (eg, city) []:New York
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Wrox
Organizational Unit Name (eg, section) []:Shell Scripting Recipes
Common Name (eg, YOUR name) []:Steve Parker
Email Address []:steve@steve-parker.org
$ cat server.crt server.key > server.pem
$ cat server.pem
-----BEGIN CERTIFICATE-----
MIID0TCCAzqgAwIBAgIJAMCjRE2qwt6vMA0GCSqGSIb3DQEBBQUAMIGiMQswCQYD
VQQGEwJVUzERMA8GA1UECBMlTmV3IFlvcmsxETAPBgNVBACTE5ldyBZb3JrMQ0w
CwYDVQQKEwRXcm94MSAwHgYDVQQLExdTaGVsbCBTY3JpcHRpbmcgUmVjaXB1czEV
MBMGA1UEAxMMU3RldmUgUGFya2VyMSUwIiwYJkoZIHvcNAQKBghZzdGV2ZUBzdGV2
ZS1wYXJrZXIub3JnMB4XDTEwMDMwMTIyMTkwN1oXDTEwMDIyNjIyMTkwN1owGAIx
CzAJBgNVBAYTA1VTMREwDwYDVQQIEwhOZXcgWW9yazERMA8GA1UEBxMlTmV3IFlv
cmsxDTALBgNVBAoTBFBdyb3gxIDAeBgNVBAsTF1NoZWxsIFNjcmlwdGluZyBSZWNP
cGVzMRUwEwYDVQQDEwxdGV2ZSBQYXJrZXIxJTAjBgkqhkiG9w0BCQEFnN0ZXZl
QHN0ZXZlLXBhcm1lci5vcmcwZz8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAK4g
TJRxAha8mEWB/fwi7vWVsGrm9p+vYtANF4MmcMftyubAeN7fYSLk0vlyaqOjWDT0
aNTfdCPZRqNm6NPGKUINu0ScTlCyarBSLMupIiliv3Y4zj3s/XFU1zZnqYECynEw
DvpoxjwnSC/fQXI04/fN9aRTuF256qsLkJLgiOJdAgMBAAGjggELMIIBBzAdBgNV
HQ4EFgQUi0uy6g40AfzIlwB1JDg8DWlI8AwgdcGA1UdIwSBzzCBZIAUIouy6g40
AfzIlwB1JDg8DWlI8ChgaikgaUwgaIxgzAJBgNVBAYTA1VTMREwDwYDVQQIEwhO
ZXcgWW9yazERMA8GA1UEBxMlTmV3IFlvcmsxDTALBgNVBAoTBFBdyb3gxIDAeBgNV
BAsTF1NoZWxsIFNjcmlwdGluZyBSZWNPcGVzMRUwEwYDVQQDEwxdGV2ZSBQYXJr
ZXIxJTAjBgkqhkiG9w0BCQEFnN0ZXZlQHN0ZXZlLXBhcm1lci5vcmeCCQDAo0RN
qsLerzAMBgNVHRMEBTADAQH/MA0GCSqGSIb3DQEBBQUAA4GBAET/0Rkhy7QLnOWW
pVrUXtnLylCg/gpsYFkLwhy5NNWOJ/d3hNMWfG2e1Ha64C/9bsPJzlp3itfhpK/g
Ff8ib2zRXctThNcmnGZbEylCF8svWus0Gj0be3+tkNn8orfFqj00Gi/JqTGD1CML
EZgOjdaIjeJA/p9uDHfjSvRnMGkx
-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4, ENCRYPTED
DEK-Info: DES-EDE3-CBC, D3A3DF183CADA55E

```

```
tA9ZEN0T6sT0wKJAGk7vzhslW3ZT5H96yDjcTFwrJ5m5mAJBNa7UxZuEeU+6vikG
ZK+X9DAfRU4MAzacSLCoGDQUcicAr43gpPqRmnSloKCKMe9/DszqpeHsXAWCX42
4/4iEsL3nctTldNWrrh90vJkNOgaw6I+4CjfxZa5OcgACQouIEjOz4CEg904c3oui
8UBWyTs/F5JI6v1RQ34r9Xl1rbj+ApJWz6poHPcsjT6L0RTDKWZx5u3VW3BS/WmU
jXFLsdrZleiK4+4aaGOqh0CC3yoMfVb4EJQliA4uYo/3NLq11J7QkC72GJWm2Q95
L2a5waHoMR6A5t8HbpfqkXEHRToNWypQCAEKhc9aR+1lrcVLml7/gqdqf+Dvc/Fx
xRcoifriuF31QiqcWRs4I/LtAPvzTmcTcWLRm4eMR+mQGK3WSVScRCoXJQ9WtLaj
aAhmDiz8tHWwP+9r2zy6dB51FJAx88h7AUe5YEPlBVQ5utgo/bVZUg/Ly7XlmmBl
Thsnqc4J92c1sEOIbrEU+kYsyu5nfWRb54PUee3jovBaSZHUPEQw128Wc0msDQBs
DFE6m/PvMTLlt1snciPZ2Dp4sVZVgXtUbIvnFIoYzH10LmerkbnvjaxEphicBQ9Or
UHu3PZksPX1RbQrW+MLKdrdzEQRBh1qToTsHViTIVsT1RbzUUdZxMzyth271AdFx
kK2fxLTbMkwHobSnPHu9TPwNkdbw8Yfmry2aFbL8FwjRLXEv5PjCkeQUZgnn51nU
vCal0016DYNCF5DZ6RrFK7wr/8atsesanzyXnIc/6OM=
-----END RSA PRIVATE KEY-----
$
```

我们现在可以启动 SSL Web 服务器了。在当前目录中创建一个文本文件，命名为 README，并写入一些祝贺的语句。这里使用的是“Success! The Shell Scripting Recipes Self-Signed Key has worked!”。设置好后，运行 `openssl s_server`，如下面的代码片段所示。遇到提示后输入密钥口令。



如果 `server.pem` 不在当前目录中，应当提供具体位置 `-cert /var/tmp/server.pem`，如下所示。在默认情况下，`openssl` 会使用 `./server.pem`。

```
$ openssl s_server -cert /var/tmp/server.pem -accept 4433 -www
Enter pass phrase for server.pem: welcome123
Using default temp DH parameters
Using default temp ECDH parameters
ACCEPT
```

现在启动 Web 浏览器，并指向 `https://localhost:4433/README`（一定要是 `https`，而不是 `http`）。我们应当会遇到一个警告，因为密钥不是浏览器所知的认证机构签署的。浏览器会显示创建密钥时输入的一些细节。这对于身份识别没有任何作用——创建密钥时可以随意输入任何内容。然而这些内容是 SSL 协议用来加密的。浏览器中的证书如图 14-4 所示。

接受该证书，浏览器会继续运行，并获取文件 README。该文件会显示在浏览器窗口中，就像其他任何文本文件的显示一样。服务器会显示连接状态，消息 `FILE:README` 说明文件 README 被请求，如图 14-5 所示。

```
$ openssl s_server -accept 4433 -www
Enter pass phrase for server.pem:
Using default temp DH parameters
Using default temp ECDH parameters
ACCEPT
8413:error:14094418:SSL routines:SSL3_READ_BYTES:tlsv1 alert unknown ca:s3_pkt.c:11
02:SSL alert number 48
```

```
8413:error:140780E5:SSL routines:SSL23_READ:ssl handshake failure:s23_lib.c:142:
ACCEPT
FILE:README
ACCEPT
ACCEPT
ACCEPT
```



图 14-4

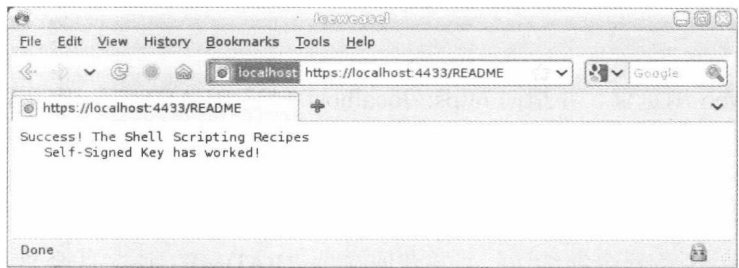


图 14-5

我们还可以使用 `openssl s_client` 测试 `openssl s_server` Web 服务器。来自 `s_server`(处于 `-WWW` 模式时)的响应是发送恰当的 HTTP 头部,接着是 `README` 文件。服务器再次显示 `ACCEPT` 与 `FILE:README` 消息,但与 Web 浏览器不同, `s_client` 不会对该测试证书是自签名的这一事实提出任何问题。它会注意到验证错误,但继续进行加密。

```
$ openssl s_client -quiet -connect localhost:4433
depth=0 /C=US/ST=New York/L=New York/O=Wrox/OU=Shell Scripting Recipes/
CN=Steve Parker/emailAddress=steve@steve-parker.org
```

```

verify error:num=18:self signed certificate ← 自签名的文件在这里被标记
verify return:1                               出来, 但加密处理继续进行。
depth=0 /C=US/ST=New York/L=New York/O=Wrox/OU=Shell Scripting Recipes/
CN=Steve Parker/emailAddress=steve@steve-parker.org
verify return:1
GET /README HTTP/1.0
HTTP/1.0 200 ok
Content-type: text/plain

Success! The Shell Scripting Recipes
    Self-Signed Key has worked!
read:errno=0
$

```

上面的例子演示了 OpenSSL 的基本功能。如果不使用 -WWW 选项, 任何向客户端输入的文本都会被服务器回显, 反之亦然。这是个极其简单的建立经过加密的 netcat 类型连接的方法。在一台服务器上, 我们可以启动 `openssl s_server` 来侦听输入的数据。在远程服务器上, 我们向前一台服务器发送文件、数据或任意什么东西。对于真正 scp 类型的安全, 我们需要加入 PAM 实现认证机制等。但 openssl 是一个简单但强大的用来双向测试 SSL 连接的工具。

下面的例子建立了一个服务器(使用 `server$` 提示符)来侦听 4433 端口, 并将接收到的任何数据写入 `/tmp/data.bin`。客户端(使用 `client$` 提示符)通过网络向服务器发送二进制文件 `/bin/ls`。

```

server$ openssl s_server -quiet -accept 4433 > /tmp/data.bin
client$ cat /bin/ls | openssl s_client -quiet -connect server1:4433
server$ ls -lh /bin/ls /tmp/data.bin ← 文件大小一样。
-rwxr-xr-x 1 root root 106K Apr 28 2010 /bin/ls
-rw-rw-r-- 1 steve steve 106K Mar 1 23:14 /tmp/data.bin
server$ md5sum /tmp/data.bin /bin/ls ← 文件的校验和相同, 经过网络
d265cc0520a9a43e5f07ccca453c94f5 /tmp/data.bin      传输时没有损坏。
d265cc0520a9a43e5f07ccca453c94f5 /bin/ls
$

```

14.10 nohup

如果我们交互式或者在 shell 脚本中运行某个进程, 希望进程运行完毕且不要因为用户登出而被关闭, 那么 `nohup` 命令会告诉 shell 在控制终端登出系统的时候忽略任何发送过来的“挂起”(HUP)信号。进程仍然可以使用 `kill -9` 来关闭。但如果网络故障导致到系统的链接断开、用于连接的客户端崩溃, 或者有各种导致连接断开的理由, 我们的进程将会继续运行。

`nohup` 主要用来在一台远程服务器上或者并行地在多台远程服务器上运行执行时间较长的命令。无论是使用自动化还是简单地使用一系列的 `ssh` 命令, 我们可以登录到机器中, 运行 `nohup /path/to/somecommand &`, 然后再登出。`&` 将命令带到后台。一般而言, 如果我

们不需要等待命令执行的结果，则可以使进程在后台运行。也就是说回到 shell 提示符，以便登出。

在默认情况下，进程需要读取的任何输入都会从/dev/null 重定向，而进程的输出与错误则会写到当前目录下的 nohup.out 文件中。如果 stdout 与 stderr 都被重定向到别处，则 nohup.out 不会被创建。下面这个脚本只是在后台生成了一个长期运行的命令，然后返回到菜单。文件数目一直保持不变，直到后台作业的结束。如果脚本直接向 \$thefile 执行写操作，则文件数目会随着时间增长，直到作业结束。如果这个会话出于某种原因被中断，find 还会继续运行。因此菜单的下一个用户可以得到最后完整的正确答案。



```
# cat menu.sh
#!/bin/bash
thefile=/var/log/filelisting.dat
tempfile=`mktemp`

select task in count recreate
do
    case $REPLY in
        1) wc -l $thefile ;;
        2) echo "Recreating the index. It will be ready in a few minutes."
            (nohup find / -print > $tempfile 2>&1 ; mv $tempfile $thefile) & ;;
        esac
    done

# ./menu.sh
1) count
2) recreate
#? 1
895992 /var/log/filelisting.dat
#? 2
Recreating the index. It will be ready in a few minutes.
#? 1
895992 /var/log/filelisting.dat
#? 1
915128 /var/log/filelisting.dat
#?
```

menu.sh

14.11 seq

seq 按顺序显示数字。它与 BASIC 编程语言有相似之处。BASIC 语言在实现 for 循环时，使用一个起始数与一个终止数，还有一个可选的步长。seq 也可以采用与之不同的 printf 类型格式，而且也会对输出自动填充，将每项数字填充到使用的最大宽度。这对固定列宽的情况很有用，如输出是 001~100 而不是 0~100。

14.11.1 整数序列

下面的脚本使用两个 `seq` 语句。外层循环使用 `seq 10 10 40` 为 `for` 循环所用。`seq 10 10 40` 从 10 计数到 40，每次增加 10。所以子网是 192.168.10.0/24、192.168.20.0/24、192.168.30.0/24 与 192.168.40.0/24，分别用于生产、备份、应用与心跳。



/24 表示前 24 位(3 字节, 像 192.168.10、192.168.20 等)是地址的网络部分。这样剩下的最后一个字节表示主机地址, 也就是地址中标识网络中特定主机的部分。

内层循环从 30 计数到 35，因为这 6 个节点在各自网络中使用相同的主机地址。它们的主机名也绑定到主机地址上，所以 `node030` 也就是 `node030-prod`、`node030-bkp`、`node030-app` 或者 `node030-hb`，具体取决于访问时使用的网络。对于生产网络，原始的名称 `node030` 也与 IP 地址关联起来。内网中简单的 `if` 语句通过在输出中添加额外的名称来关联到具体主机。



```
$ cat hosts.sh
```

```
#!/bin/bash
```

```
for subnet in `seq 10 10 40`
do
```

```
  case $subnet in
```

```
    10) suffix=prod
        description=Production ;;
```

```
    20) suffix=bpk
        description=Backup ;;
```

```
    30) suffix=app
        description=Application ;;
```

```
    40) suffix=hb
        description=Heartbeat ;;
```

```
  esac
```

```
  cat - << EOF > /tmp/hosts.$subnet
```

```
# Subnet 192.168.${subnet}.0/24
```

```
# This is the $description subnet.
```

```
EOF
```

```
  for address in `seq 30 35`
```

```
  do
```

```
    # For Production network, also add the raw node name
```

```
    if [ "$suffix" == "prod" ]; then
```

```
      printf "192.168.%d.%d\tnode%03d\tnode%03d-%s\n" \
```

```
        $subnet $address $address $address $suffix >> /tmp/hosts.$subnet
```

```
    else
```

```
      printf "192.168.%d.%d\tnode%03d-%s\n" \
```

```
        $subnet $address $address $suffix >> /tmp/hosts.$subnet
```

```
        fi
    done
    cat /tmp/hosts.$subnet
done

$ ./hosts.sh

# Subnet 192.168.10.0/24
# This is the Production subnet.
192.168.10.30 node030 node030-prod
192.168.10.31 node031 node031-prod
192.168.10.32 node032 node032-prod
192.168.10.33 node033 node033-prod
192.168.10.34 node034 node034-prod
192.168.10.35 node035 node035-prod

# Subnet 192.168.20.0/24
# This is the Backup subnet.
192.168.20.30 node030-bkp
192.168.20.31 node031-bkp
192.168.20.32 node032-bkp
192.168.20.33 node033-bkp
192.168.20.34 node034-bkp
192.168.20.35 node035-bkp

# Subnet 192.168.30.0/24
# This is the Application subnet.
192.168.30.30 node030-app
192.168.30.31 node031-app
192.168.30.32 node032-app
192.168.30.33 node033-app
192.168.30.34 node034-app
192.168.30.35 node035-app

# Subnet 192.168.40.0/24
# This is the Heartbeat subnet.
192.168.40.30 node030-hb
192.168.40.31 node031-hb
192.168.40.32 node032-hb
192.168.40.33 node033-hb
192.168.40.34 node034-hb
192.168.40.35 node035-hb
$
```

hosts.sh

如果要直接写到/etc/hosts 中, 上面的脚本可以像./hosts.sh >> /etc/hosts 这样调用, 或者将循环中的 cat 语句追加写入/etc/hosts 中。该脚本也可以使用一个开关, 用来提供追加(或写入)的文件名。上面这种方式更具灵活性, 因为输出可以写向任何文件(或不写到任何

文件，只显示到 `stdout`）。

14.11.2 浮点数字列

`seq` 的应用不仅仅局限在整数。下面这个简单的脚本将英里转换为公里，并且小数部分也显示了出来。



可从
wrox.com
下载源代码

```
$ cat miles.sh
#!/bin/bash
# 1m ~= 1.609 km

for miles in `seq 1 0.25 5`
do
    km=`echo "scale=2 ; $miles * 1.609" | bc`
    printf "%0.2f miles is %0.2f kilometers\n" $miles $km
    #echo "$miles miles is $km km"
done
```

```
$ ./miles.sh
1.00 miles is 1.61 kilometers
1.25 miles is 2.01 kilometers
1.50 miles is 2.41 kilometers
1.75 miles is 2.82 kilometers
2.00 miles is 3.22 kilometers
2.25 miles is 3.62 kilometers
2.50 miles is 4.02 kilometers
2.75 miles is 4.42 kilometers
3.00 miles is 4.83 kilometers
3.25 miles is 5.23 kilometers
3.50 miles is 5.63 kilometers
3.75 miles is 6.03 kilometers
4.00 miles is 6.44 kilometers
4.25 miles is 6.84 kilometers
4.50 miles is 7.24 kilometers
4.75 miles is 7.64 kilometers
5.00 miles is 8.05 kilometers
$
```

miles.sh

14.12 sleep

本书广泛使用 `sleep`，它与 `date` 的组合可以用来提供宝贵的调试信息。`sleep` 的 GNU 实现还可以接受十进制分数，以及使用后缀 `m`、`h` 与 `d` 分别表示分钟、小时与天。通过在脚本中插入 `sleep` 语句，我们可以有效地在插入位置暂停执行，然后查看运行情况。本章后面的 14.13 节充分利用 `sleep` 命令模拟了应用程序关闭脚本的不同场景。

另外 `sleep` 也经常用在循环中。如果要每分钟执行一组命令，简单的 `sleep 60` 则比在

cron 中对任务进行调度要容易一些。使用 cron 每 90 秒钟运行某个任务则要更加困难，但 sleep 语句非常适合这样的任务。



```
$ cat memory.sh
#!/bin/bash
LOGFILE=/var/tmp/memory.txt
while :
do
    RAM=`grep MemFree /proc/meminfo | awk '{ print $2 }'`
    echo "At `date +%H:%M on %d %b %Y` there is $RAM Kb free on `hostname -s`" \
        |tee -a $LOGFILE
    sleep 60
done
```

```
$ ./memory.sh
At 12:45 on 25 Mar 2011 there is 500896 Kb free on goldie
At 12:46 on 25 Mar 2011 there is 441336 Kb free on goldie
At 12:47 on 25 Mar 2011 there is 213736 Kb free on goldie
At 12:48 on 25 Mar 2011 there is 82936 Kb free on goldie
At 12:49 on 25 Mar 2011 there is 96996 Kb free on goldie
At 12:50 on 25 Mar 2011 there is 87240 Kb free on goldie
At 12:51 on 25 Mar 2011 there is 493826 Kb free on goldie
```

memory.sh

人们通常低估 sleep 的作用。虽然它属于那种小而且似乎不太重要的工具，但它的作用是不可或缺的。sleep 的 GNU 扩展使它更易于使用。sleep 1h 的可读性比 sleep 3600 更好。sleep 2d 则比 sleep 172800 容易理解得多。sleep 中小数秒的功能也许不是非常有用，因为如果没有实时操作系统，sleep 唯一能保证的是它不会比请求的时间返回得更早(除非 sleep 进程被关闭)。本书后面的实用脚本 17-1 利用次秒级的 sleep 命令让游戏的进度越来越快。

14.13 timeout

read 与 select 命令的功能要归功于 TMOUT 变量，因为 TMOUT 变量定义了这两个命令等待交互式输入的最大秒数。其他的命令没有将这一功能嵌入其中。但这是一个很重要的功能，尤其是对于脚本运行。下面这个简单的脚本对于演示 timeout 的作用很有用，因为它不总是按照我们预期的方式运行。



```
$ cat longcmd.sh
#!/bin/bash
trap 'echo "`date`: ouch!'" 15
echo "`date`: Starting"
sleep 20
```

```
echo "`date`: Stage Two"
sleep 20
echo "`date`: Finished"
```

longcmd.sh

在第一次运行时, `-s 15 3` 告诉 `timeout` 在 3 秒钟之后向脚本发送一个 `SIGTERM`(15 号信号)。该信号被脚本捕获, 但它产生的效果是终止第一个 `sleep` 命令。所以在 3 秒钟之后的 13:33:46, 脚本处理了这个捕获的信号, 并显示“ouch!”消息。执行从脚本中的下一个命令开始恢复, 回显日期(还是 13:33:46), 然后休眠 20 秒, 最后在 13:34:06 运行完毕。

```
$ timeout -s 15 3 ./longcmd.sh ; date
Thu Mar 24 13:33:43 GMT 2011: Starting
Terminated
Thu Mar 24 13:33:46 GMT 2011: ouch!
Thu Mar 24 13:33:46 GMT 2011: Stage Two
Thu Mar 24 13:34:06 GMT 2011: Finished
Thu Mar 24 13:34:06 GMT 2011
```



GNU 中的 `coreutils` 到第 7 个版本才有 `timeout` 工具。之前, 很多发行版中包含的 `timeout` 工具来自 The Coroner's Toolkit(<http://www.porcupine.org/forensics/tct.html>)。其中的 `timeout` 有着不同的语法, 并且输出更详细。它没有 `-s` 标志, 所以使用 `-15 3` 表示 `-s 15 3`。它完全没有 `-k` 选项。从 RHEL6、Debian 6、Ubuntu 11.04 与 SuSE 11 开始包含 GNU `coreutils` 版本的 `timeout`。

如果脚本不捕获 `SIGTERM`, 则在接收到信号时会立即终止运行。`timeout` 在处理不能顺利退出的顽固代码时特别有用。

```
$ timeout -s 15 3 ./longcmd-notrap.sh ; date
Thu Mar 24 20:12:45 GMT 2011: Starting
Thu Mar 24 20:12:48 GMT 2011
$
```

添加 `-k 12` 开关告诉 `timeout` 在初始的 `SIGTERM` 之后 12 秒向进程发送 `SIGKILL` 信号。“ouch!”消息由于 `SIGTERM` 信号在 3 秒之后再次显示出来。再经过 12 秒之后, 整个脚本被关闭。脚本没有完成 20 秒钟的休眠, 也没有显示休眠之后的 `Finished` 消息。这种处理超时的方式更具强制性。

```
$ timeout -s 15 -k 12 3 ./longcmd.sh ; date
Thu Mar 24 13:34:09 GMT 2011: Starting
Terminated
Thu Mar 24 13:34:12 GMT 2011: ouch!
Thu Mar 24 13:34:12 GMT 2011: Stage Two
Killed
Thu Mar 24 13:34:24 GMT 2011
```

同样，第一个信号可以是 SIGKILL，方法是将 -s 标志指定为 9(或 KILL)。这样做的结果是一旦到达指定的超时时间就立即关闭进程。

```
$ timeout -s 9 3 ./longcmd.sh ; date
Thu Mar 24 13:34:35 GMT 2011: Starting
Killed
Thu Mar 24 13:34:38 GMT 2011
$
```

timeout 的一个实用用法是调用难以控制的应用程序。这些不好控制的程序要么编写得很糟糕，要么依赖于一些无法控制的外部因素。举一个前者的例子，在为应用程序编写关闭脚本时发现这些程序会挂起，无法整齐地退出。而后者的例子便是，在外部网络服务器上的一个下载任务挂起或者不能如愿完成。假定我们不需要编写任何复杂的结构来处理这些情况，使用 timeout 处理则非常合适。接下来两节处理了这两种情况，并演示了如何使用 timeout 提供更具可管理性的服务。

14.13.1 关闭脚本

关闭脚本可以使用 timeout 包装处理难以处理的进程。它提供一个关闭过程可耗费的明确、已知的最大时间。这里的关闭程序/usr/local/bin/stop.myapp 使用了 50 秒来完成(返回码为 20)。如果它捕获到某个信号，则会在以返回码 20 退出之前休眠 20 秒。



如果超时(20 秒以后)，程序会以返回码 124 退出。如果程序被关闭(20 加 10 秒之后)，则会以返回码 139 退出。因为 stop.myapp 程序被设计成在关闭应用程序时不给出任何信息，所以如果它失败了，myapp.sh 初始化脚本会使用存储在/var/run/myapp.pid 中的 PID 强制关闭没有关闭的应用程序。应用程序无法回避这一信号。但是到这一步，初始化脚本已经尽其所能让系统不残留任何应用程序，可以干干净净地关闭。



timeout 手册页中描述，如果用 timeout 执行的命令超时，则命令会以返回码 124 退出。然而，如果该命令超时且必须用 SIGKILL(9)关闭，则 timeout 命令本身也会终止，并以返回码 137 退出(128 加上关闭时发送给它的信号值)。因此，检测 137 比检测 124 更有用。

像这样的应用程序可能很难确定，所以下面的脚本还会在每次关闭时将返回码记录到/var/log/myapp.log 中，以便记录超时的频率。另外要注意的是，永远也不会执行到 exit 10。一旦运行到代码的这一部分，则表明已经超时，所以 timeout 会返回 124 或 139。返回值取决于它是否要关闭程序。



可从
wrox.com
下载源代码

```
$ cat /etc/init.d/myapp.sh
```

```
#!/bin/bash
```

```
function killapp
```

```
{
    # if we get here, the application refused to shut down.
    kill -9 `cat /var/run/myapp.pid`
}
```

```
case $1 in
```

```
start)
```

```
    echo "Starting myap..."
    /usr/local/bin/myapp &
    echo $! > /var/run/myapp.pid
    ;;
```

```
stop)
```

```
    echo "Stopping myapp..."
    timeout -s 15 -k 10 20 /usr/local/bin/stop.myapp
    res=$?
    echo "`date`: myapp returned with exit code $res" >> /var/log/myapp.log
    case "$res" in
        0) echo "NOTE: myapp stopped by itself." ;;
        124) echo "NOTE: myapp timed out when stopping."
            killapp ;;
        137) echo "NOTE: myapp was killed when timing out."
            killapp ;;
        *) echo "Note: myapp exited with return code $res" ;;
    esac
    rm -f /var/run/myapp.pid
    ;;
```

```
*)
```

```
    echo "Usage: `basename $0` start | stop"
    exit 2
```

```
esac
```

myapp.sh



可从
wrox.com
下载源代码

```
$ cat /usr/local/bin/stop.myapp
```

```
#!/bin/bash
```

```
trap may_die 1 3 9 15
```

```
function may_die
```

```
{
    SLEEP=`expr $RANDOM % 20`
    echo "Sleeping for $SLEEP seconds (but you don't know that)"
    sleep $SLEEP && exit 10
}
```

```
TIME=`expr $RANDOM % 50`
```

```
echo "STOPPING MYAPP. (likely to take $TIME seconds, but you don't know that!)"
```



```

for i in `seq 1 $TIME`
do
    echo -en "."
    sleep 1
done

exit 20
$

```

stop.myapp

18 秒的关闭过程不受 `timeout` 的影响:

```

$ ./myapp.sh stop
Stopping myapp...
STOPPING MYAPP. (likely to take 18 seconds, but you don't know that!)
.....Note: myapp exited with return code 20

```

44 秒的关闭过程肯定会超时。因为进一步的关闭会超过允许的 10 秒，所以关闭程序被终止:

```

$ ./myapp.sh stop
Stopping myapp...
STOPPING MYAPP. (likely to take 44 seconds, but you don't know that!)
.....Terminated
Sleeping for 11 seconds (but you don't know that)
./myapp.sh: line 3: 6071 Killed          timeout -s 15 -k 10 20 /usr/local
/bin/stop.myapp
NOTE: myapp was killed when timing out.

```

尽管 `timeout` 最后紧跟了一个结尾 `exit`，但还是会超时，但它没有被关闭。

```

$ ./myapp.sh stop
Stopping myapp...
STOPPING MYAPP. (likely to take 26 seconds, but you don't know that!)
.....Terminated
Sleeping for 2 seconds (but you don't know that)
NOTE: myapp timed out when stopping.
$

```

14.13.2 网络超时

这一节要介绍的第二个例子是关于某个系统或进程，它完全取决于系统外的因素，如在远程服务器上进行的下载任务。如果服务器或者连接到服务器的网络很慢，脚本本身无济于事，但它可以使用 `timeout` 来管理运行状态。例如，如果预期的下载传输时间是 10 秒，`timeout` 可以用来在超过 1 分钟还没有完成的情况下中断文件传输。



```
$ cat downloader.sh
```

```
#!/bin/bash
```

```
for file in file1.zip file2.zip file3.zip
do
    timeout -s 9 60 wget http://unreliable.example.com/${file}
    if [ "$?" -ne "0" ]; then
        echo "An error occurred when downloading $file"
    fi
done
```

downloader.sh

```
$ ./downloader.sh
```

```
--2011-03-25 13:06:58-- http://unreliable.example.com/file1.zip
Resolving unreliable.example.com... 192.0.32.10
Connecting to unreliable.example.com|192.0.32.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 84263304 (80M) [application/zip]
Saving to: `unreliable.example.com/file1.zip'

100%[=====>] 84,263,304 9.8M/s in 8.0s
```

```
2011-03-25 13:07:06 (9.8 MB/s) - `unreliable.example.com/file1.zip' saved [84263304/84263304]
```

```
--2011-03-25 13:07:06-- http://unreliable.example.com/file2.zip
Resolving unreliable.example.com... 192.0.32.10
Connecting to unreliable.example.com|192.0.32.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 413396910 (394M) [application/zip]
Saving to: `unreliable.example.com/file2.zip'
```

```
59% [=====>] 245,297,152 36.9M/s eta 6s ./downloader.sh: line 3: 3482 Killed timeout -s 9 60 wget http://unreliable.example.com/${file}
```

```
An error occurred when downloading file2.zip
```

```
--2011-03-25 13:08:07-- http://unreliable.example.com/file3.zip
Resolving unreliable.example.com... 192.0.32.10
Connecting to unreliable.example.com|192.0.32.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 701084434 (669M) [application/zip]
Saving to: `unreliable.example.com/file3.zip'
```

```
17% [=====>] 121,831,424 22.0M/s eta 51s ./downloader.sh: line 3: 3484 Killed timeout -s 9 60 wget http://unreliable.example.com/${file}
```

```
An error occurred when downloading file3.zip
```

```
$
```

14.14 uname

uname 与 hostname 有一定联系，但前者更灵活。在 x86 架构下，uname 提供的信息比在专门的硬件供应商设计的硬件上要少一些。SunFire E25k 通过 uname -i 输出平台信息 SUNW,Enterprise-25000，T5240 输出 SUNW,T5240。uname 本身输出运行中内核的基本名称，如 Linux、SunOS 或者 FreeBSD。利用其他开关，uname 可以输出主机名(-n)、内核发行号(-r)与版本号(-v)、CPU 架构(-m)、操作系统(-o)、处理器(-i)以及硬件平台(-i)。这些信息可以使用-a 开关组合起来，与命令 uname -snrvmpio 等价。

脚本可以使用 uname 来找到运行的平台，并使自身适合于运行在该平台上。下面的脚本判断 CPU 的处理能力，并根据情况给出报告。本章之前的 14.3 节也使用了 uname 来判断运行的 OS，并假设了所使用的包管理系统种类。



```
$ cat uname.sh
#!/bin/sh
case `uname -m` in
    amd64|x86_64) bits=64 ;;
    i386|i586|i686) bits=32 ;;
    *) bits=unknown ;;
esac
echo "You have a ${bits}-bit machine."
```

uname.sh

```
$ ./uname.sh
You have a 64-bit machine.
$
```

我们最好是要知道 uname 的可能输出。表 14-1 给出了一些不同操作系统与架构情况下的 uname 输出。表中的前 3 项是不同发行版的 Linux，接下来两个都是 Solaris SPARC，然后是 x86 架构的 Solaris，最后一个是 OpenBSD 服务器。

表 14-1 uname 在不同操作系统中的输出

OS	uname -s	uname -n	uname -r	uname -m
RedHat 6	Linux	hattie	2.6.32-71.el6.i686	i686
Debian 5	Linux	goldie	2.6.32-5-amd64	x86_64
Ubuntu 10.10	Linux	elvis	2.6.35-25-generic	x86_64
Solaris 10	SunOS	db9	Generic_142900-02	sun4u
Solaris 10	SunOS	webapp	Generic_137137-09	sun4v
Solaris 10x86	SunOS	appserver	Generic_142901-02	i86pc
OpenBSD 4.8	OpenBSD	saga	4.8	i386

有些讽刺的是, `uname` 命令本身在不同的架构与操作系统中就表现出很强的非一致性。Solaris、SCO 与其他操作系统都有 `-X` 选项, 显示的信息也几乎相同, 另外还有 CPU 数目、总线架构以及其他信息。BSD 则根本没有 `-i` 选项, 而它的 GNU 实现在 x86 架构下的 `-i` 选项输出没有多大意义。

14.15 uuencode

作为 `sharutils` 包的一部分, `uuencode` 对二进制文件进行编码, 使其适合用电子邮件附件传输。因为电子邮件是基于文本的协议, 某些二进制字符会对电子邮件本身造成干扰, 所以将二进制文件转换成 7 位安全文本编码可以保证电子邮件能对文件进行处理。收件人的电子邮件客户端应当检测格式, 并显示这样的附件, 而不是把它当成电子邮件文本的一部分。`uuencode` 有些特殊, 因为尽管它可以从 `stdin` 或文件读取(以文件名作为参数传递给 `uuencode` 时), 它仍然假设会从 `stdin` 接收数据。传递给 `uuencode` 的最后一个(或者本例中唯一一个)文件名作为附件的文件名。这在处理 `stdin` 时是有意义的。文件没有名称, 但收件人需要名称来保存或打开文件。因此, 在处理文件时, 通常最好两次给出文件名。第一个是文件的名称, 第二个是收件人接收到的文件名称。头部如下例所示。前 3 个单词是 `begin`, 然后是文件的八进制权限(在电子邮件附件中不常用), 最后是呈现给收件人的文件名。在本例中, 本地文件 `sample.odt` 会以 `recipient.odt` 文件的名义发送出去。

```
$ uuencode sample.odt recipient.odt | head
begin 664 recipient.odt
M4$!#!!0``@``$&3<SY>QC(,)P``"<````(````;6EM971Y<&5A<'!L:6-A
M=&EO;B]V;F0N;V%S:7,N;W!E;F108W5M96YT+G1E>'102P,$%``("`@`09-S
M/@````````````````L``!C;VYT96YT+GAM;*5778_B-A1}[Z^(LM*^&0)#
M5UK2@56KJFJEF:K:H=6^>FP'K/57;4/@W_?: (28P9"85$@+%)P/GWM\`1Z^
M[]4L=MQY8?2BG$[NRH]K:IC0ZT7Y]^HW]+G\LOSAP=2UH+QBAFX5UP%1HP/\
M%N"M?=5:%^76Z<H0+WREB>*^"KORENO.J^JCJY2K7?'A($>[)W#?._!]&.L<
ML6>^Y&5\Y@3N>S-'FK'$.0ND]MUK,]9Y[R6J#;"N+`GBHHJ]% /K[HMR$8"N,
MFZ:9-+.)<6L\G<_G.%ESP33C[-; )A&(4<\EC,H^GDRGNL(H',K: ^B.V7I+?J
MA;O1U)!`7G75[:C%;%;#U!#-\2-UD8"G[=WQL:W=\;ZOHJ$S4!//N,G,* :O
```

LibreOffice 文档中不可控的二进制数据现在被安全地编码为可打印的(尽管人类无法阅读)7 位文本。该文本可以附加到电子邮件的主体, 然后安全地发送给收件人。在本例中, 我们通过显示文档消息 “Here is the document you wanted. Regards, Steve” 并在子 shell 中运行 `uuencode` 将文档发送给自己。子 shell 的输出以单一文本流的形式通过管道传递给 `mailx` 命令。

```
sender$ ( echo "Here is the document you wanted."; \
> echo "Regards, Steve."; \
> uuencode sample.odt mydocument.odt ) | \
> mailx -s "Document attached" steve@steve-parker.org
sender$
```



将计算机配置成可以发送电子邮件通常与配置 `DSmailhost.example.com` 一样简单(其中 `mailhost.example.com` 是内部电子邮件服务器的名称)。

如果本地计算机经过恰当的配置可以发送电子邮件,那么该消息将被发送到命令行中指定的收件人。由于电子邮件的特性,只要语法正确 `mailx` 命令就会返回成功,它无法检测到发送电子邮件中出现的任何问题。因此我们需要自行检查。收到的电子邮件如图 14-6 所示。

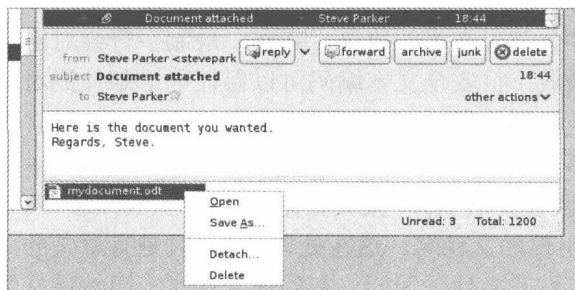


图 14-6

14.16 xargs

如果调用命令时使用了很多参数(通常都是文件名),则内核必须将这些数据放置到一个命令可以引用的位置。有时(通常是在使用 `find` 命令搜寻大型文件系统时),这样的参数列表会大过内核允许的程度。于是会导致输出如“`rm: Argument list too long`”这样的消息。很多人都相当适应这一点,并且知道使用 `xargs` 可以避免这一问题,但并不是每个人都理解其工作原理,其中有一些重要的关联。

```
$ rm `find . -name "*.core*" -print`
bash: /bin/rm: Argument list too long
$ find . -name "*.core*" -print0 | xargs -0 rm
$
```

要避免这样的问题, `xargs` 会读取其标准输入(没有参数列表过长这样的问题),然后将它们以可管理的参数块的形式传递给对应的命令。下面这个简单的 `listfiles` 脚本演示了 `xargs` 在运行中命令上的效果。本例中, `-L 3` 选项让 `xargs` 对输入进行拆分,每次调用 3 个文件。这与一般认为的 `xargs` 工作方式不同,因此结果可能会出人意料。



可从
wrox.com
下载源代码

```
$ cat listfiles
#!/bin/bash
echo "Listfiles (PID $$) was called with $# arguments:"
i=1
while [ -a "$1" ]
do
```

```

    echo "${i}: $1"
    ((i++))
    shift
done

```

listfiles

```

$ find . -print
.
./etc
./etc/hosts
./etc/config.txt
./bin
./bin/ls
./sh
./listfiles
$ find . -print | xargs -L 3 ./listfiles
Listfiles (PID 17088) was called with 3 arguments:
1: .
2: ./etc
3: ./etc/hosts
Listfiles (PID 17089) was called with 3 arguments:
1: ./etc/config.txt
2: ./bin
3: ./bin/ls
Listfiles (PID 17090) was called with 2 arguments:
1: ./sh
2: ./listfiles
$

```

find 命令的前 3 个结果作为 3 个参数传递给 listfiles:

1. .
2. ./etc
3. ./etc/hosts

接下来的 3 个文件被传递给另一个 listfiles 实例。该实例明显与之前的不同，因为它的 PID 不相同。最后只剩下两个文件，所以它们被传递给最后一次 listfiles 调用。

以上这些都没有问题，直到文件名开始包含奇怪的字符，如空格符。将 My Photos 与 My Documents 文件夹添加到目录中会对 xargs 造成极大的混淆。第一个实例调用时使用了下面 3 个参数：

1. .
2. ./My Photos
3. ./My Photos\DCIM0001.jpg

然而，脚本会将它们解释成 5 个不同的单词：

1. .
2. ./My
3. Photos
4. ./My
5. Photos\DCIM0001.jpg

在这些文件中，只有第一个是真实的文件(刚好是一个目录)。其他 4 个都通不过[-a "\$1"]测试，所以根本不会显示输出。这样的混淆会一直持续下去，直到遇到不包含空格的路径。

```
$ find . -print
.
./My Photos
./My Photos/DCIM0001.jpg
./My Photos/DCIM0002.jpg
./My Documents
./My Documents/doc1.doc
./My Documents/cv.odt
./etc
./etc/hosts
./etc/config.txt
./bin
./bin/ls
./sh
./listfiles
$ find . -print | xargs -L 3 ./listfiles
Listfiles (PID 17096) was called with 5 arguments:
1: .
Listfiles (PID 17097) was called with 6 arguments:
Listfiles (PID 17098) was called with 4 arguments:
Listfiles (PID 17099) was called with 3 arguments:
1: ./etc/config.txt
2: ./bin
3: ./bin/ls
Listfiles (PID 17100) was called with 2 arguments:
1: ./sh
2: ./listfiles
$
```

为了避免这样的问题，xargs -0 会用 ASCII 的 0 字符来分离文件名，并将各种其他空白字符当成文件名的一部分。find -print0 与 locate -0 语法针对各自的命令也都支持这种列出文件名的方法。这样做更具健壮性，并且默认就应当如此。

```
$ find . -print0 | xargs -0 -L 3 ./listfiles
Listfiles (PID 17129) was called with 3 arguments:
1: .
2: ./My Photos
3: ./My Photos/DCIM0001.jpg
```

```

Listfiles (PID 17130) was called with 3 arguments:
1: ./My Photos/DCIM0002.jpg
2: ./My Documents
3: ./My Documents/doc1.doc
Listfiles (PID 17131) was called with 3 arguments:
1: ./My Documents/cv.odt
2: ./etc
3: ./etc/hosts
Listfiles (PID 17132) was called with 3 arguments:
1: ./etc/config.txt
2: ./bin
3: ./bin/ls
Listfiles (PID 17133) was called with 2 arguments:
1: ./sh
2: ./listfiles
$

```

该命令将所有的参数传递给 `xargs`，但有些命令希望文件名以不同的方式传递给它们。如果我们要搜索每个在日志文件中找到的文件，则应当使用 `grep -nw "^${filename}$" /tmp/interesting` 命令。该命令只会搜索整个文件名的精确匹配。然而，该命令的行为并不十分吻合我们的要求：

```
find . -print0 | xargs -0 grep -n /tmp/interestingfiles
```

当然，文件名需要处于 `-n` 与 `/tmp/interestingfile` 之间。文件名的标准占位符是 `{}`，但也可以使用 `-I` 标志进行修改。该命令行将文件名放到 `grep` 命令参数的适当部分。`xargs` 也非常智能地从 `-I` 进行推断，让每个 `grep` 实例只调用一个文件名进行搜索。

```

$ cat /tmp/interestingfiles
./bin/ls
./My Documents/cv.odt
./usr/bin/sleep
$ find . -print0 | xargs -0 -I{} grep -nw "^{}$" /tmp/interestingfiles
2:./My Documents/cv.odt
1:./bin/ls
$

```

14.17 yes

很多实用程序(如 `fsck`)都有一个 `-y` 选项用来对程序提出的所有问题回答 `yes`。还有一些较好的实用程序，它们对任何问题的 `yes` 回答基本上都表示相同的意思——无论是要标记超级块为 `clean` 还是删除 `inode`，`fsck` 的 `-y` 标志表示程序应当尽其所能地修复文件系统。这在脚本无须与用户进行交互时非常有用。有一些工具没有这个选项，于是 `yes` 可以用于交互式使用。`yes` 所做的就是不断地提供字母 `y`。下面这两个命令是等价的。

```

# fsck -y /dev/sdf1
# yes | fsck /dev/sdf1

```




作为一个可选方案，yes 可以接受任何其他文本作为参数并显示它们。

下面这一行命令对 yes 的用法或许不是非常明显：

```
$ yes no | cp -i * /tmp
```

该命令用管道将 no 传递给使用 -i 选项的 cp 命令，该选项会询问是否覆盖文件。该命令的效果就是不覆盖已有文件，但其他文件会被复制到/tmp。这种需求不是很常见，但看起来至少比较为用户考虑，偶尔也会用到。

14.18 本章小结

系统管理需要掌握一组复杂且相互关联的工具。希望本章介绍的内容能够向大家展示如何在 shell 脚本中使用这些工具将常见的系统管理任务自动化，使得复杂的任务变得简单、快捷且可重复执行，还能让基本操作更灵活、更有用。

本书的第 I 部分更多地介绍关于理论、概念以及 Unix 和 GNU/Linux 生态系统中特殊工具与功能的工作原理的细节。本书剩下的部分由一些实用脚本组成，主要集中在任务本身以及完成任务的方式与使用这种方式的原因。这些脚本基于之前介绍的知识与信息，用来构建可靠、实用的 shell 脚本(可以照搬使用或者修改后用于特定用途)。这些脚本还提供了一些有用的例子，它们将 shell 工具与功能组合起来用于各种目的。

第Ⅲ部分

系统管理的实用脚本

- 第 15 章 shell 特性
- 第 16 章 系统管理
- 第 17 章 演示
- 第 18 章 数据存储与检索
- 第 19 章 数值
- 第 20 章 进程
- 第 21 章 国际化

第 15 章

shell 特性

本章介绍 3 个特定任务：为任意发行版的 Linux 或 Unix 安装初始化脚本、报告已安装的 RPM 包的信息，以及 Kickstart 的 postinstall 脚本。

第一个实用脚本给出了一些有关可移植性的技术。其中最重要的部分可能是 15.1.4 节描述的四步分解法。尽管在判断出发行版后将脚本复制到相应的位置这一做法看起来更加有效，但如果能意识到任务中存在独立子任务并且一次只完成一步，这样的代码则可维护性更强。

RPM 报告脚本利用第 9 章中的数组与第 7 章中一些更加复杂的变量结构编写了一个非常简洁的脚本来实现一个常规任务。另外它还演示了如何在 shell 脚本中进行简单的 HTML 格式化，以便更好地利用图形化 Web 浏览器的显示功能。这样的演示效果要比纯文本输出看起来更加专业。

Kickstart 的 postinstall 脚本调用了函数库，而且还利用 here 文档与重定向创建了一个简单且可维护的 Kickstart 环境。另外还演示了不同形式的条件执行如何用来匹配不同的情形。有时对 \$? 中的返回码进行测试可以使代码更美观，有时 [command] && command 的可读性更好。很多时候都是个人喜好问题，但经过一段时间并有了经验之后，我们就会为表达式选择更直观的语法。

15.1 实用脚本 15-1：安装初始化脚本

这里使用了条件执行——if、test 和 case——来判断在计算机启动时如何自动打开进程。如今大多数 GNU/Linux 系统都将初始化脚本安装到/etc/init.d，但也不完全如此。实际的注册(如果用到的话)与打开进程的方法因发行版而异。第 16 章的实用脚本 16-1 会对初始化脚本本身进行介绍。

15.1.1 用到的技术

- if、else、elif

- case
- [expression] && command

15.1.2 概念

初始化脚本是一个比较简单的 shell 脚本。它在系统启动时开启服务，在系统关闭时关闭服务。本脚本可以用于通用软件安装例程，在软件不依靠包管理系统安装时自动安装初始化脚本。<http://lwn.net/Distributions> 列出了超过 500 种的不同发行版。大多数发行版都属于几种主要类别(基于 Red Hat、基于 Debian 等)中的一种，但有些则比较独特。

15.1.3 潜在的陷阱

脚本本身有一些陷阱。最坏的情况是失败时无法识别这些失败并向用户报告。该脚本确实对运行所在的发行版进行试探，虽然没有成功，但还是显示了所做的工作。按照一般的 Unix 传统，它是一个静默脚本。在成功运行时完全保持静默状态，只有在向用户发出警告或者出错时才会显示消息。

另一个较为特殊的陷阱是对某个发行版给出错误的要求。这种陷阱避免起来不会总是像听起来那样简单，因为发行版多种多样，并且任意发行版都有可能按照各自的喜好变更其工作原理。越是大型与稳定的发行版进行任意变更的可能性就越小，但 Upstart 与 SMF 分别是 Ubuntu 与 Solaris 中对系统启动机制制作的较为显著的变更，而系统启动机制在近几十年来一直都处于非常稳定的状态。本脚本特定的设计结构旨在防止这类随着岁月的变迁而在脚本中慢慢出现的问题。

15.1.4 脚本结构

像这样的脚本结构对于日后脚本的运行与维护都有着强烈的影响。因为该脚本的作用实在无足轻重，所以它并不会像软件中负责启动的模块一样得到相同程度的细节与质量控制。这甚至都不是初始化脚本，它只是一个用来安装初始化脚本的一次性脚本。很明显的问题是“为什么要考虑到质量”。原因在于随着时间的推移，脚本会变得越来越难看且难以维护，并很快变得无法控制且充满错误。这样的脚本在添加对新发行版的支持时会不经意地破坏对另一个发行版的支持，并且对代码的修复还会破坏对其他发行版的支持。从一开始就使用正确的结构会使得脚本在将来具有灵活性与可移植性，并且新的开发人员在修改代码时就能知道他们应当在何处做些什么。我们很快就能想到去检测 Red Hat(打个比方)，然后立刻将初始化脚本复制到/etc/init.d 中，运行 chkconfig，最后结束。这对于 Red Hat 是可行的，但它忽略了也需要脚本处理的更复杂的细微之处。

该脚本分为 4 个不同的步骤：

- (1) 判断发行版。
- (2) 安装到正确的初始化目录。
- (3) 注册服务。
- (4) 启动服务。

1. 判断发行版

大多数发行版都提供了方便地判断所属系统的方法。通常这是/etc 中的一个文件，可以用来确定发行版。文件的内容一般会提供一些更细节的信息。例如，我们依赖于 SuSE 10.0 或更高版本的某一特性，通过编程对/etc/SuSE-release 分析可以得知这一特性是否可用。本步骤所做的仅仅是确定使用的发行版。其他与发行版相关的工作不在这一阶段进行，而是在步骤(2)中。这样使得脚本为这一任务将类似的发行版绑定在一起。

在确定发行版时 if / elif / else 结构是最理想的，因为每个发行版都要执行不同的操作，无论这些操作是检查是否存在某个特定文件或者执行其他一些类似的处理任务。最后的 else 语句会设置 distro=unknown，用于随后的处理。

2. 安装到初始化目录

确定了发行版之后，第二个步骤会基于发行版将文件安装到正确的目录下。对于很多现代的 GNU/Linux 发行版而言，简单的一条 init_dir=/etc/init.d 语句就足够了，但其他一些发行版则不太一样。

该步骤使用 case 结构遍历所有可能不同的发行版。它还使用了表示包含一切的*子句，对于未知发行版将循环遍历各种常见目录。如果发现这些目录中有某个存在，则假设该目录就是安装初始化脚本的正确位置。这使得子句可以显示一些可能有用的判断力。需要用户知晓的一个重点是这里的操作有一些猜测的意味存在，因为如果猜错，用户会得到一些关于问题出在何处的提示。

3. 注册服务

在启动之前，服务必须向系统进行注册。按照惯例，注册是指将/etc/init.d 中的符号链接安装至 rcN.d 中，这里的 N 表示改后的运行级别。有些链接名称为 SXXservice，S 表示启动(Start)，XX 表示服务器启动的顺序，从最低到最高。这些脚本调用时都会带单个 start 参数。其他的链接名称为 KXXservice，K 表示关闭(Kill)。重启计算机或者改变运行级别时，对它们的调用会使用单个 stop 参数，使服务完全关闭。一些近期的 GNU/Linux 发行版则是使用 chkconfig。该程序分析初始化脚本中的注释来决定使用何种运行级别与启动顺序。

4. 启动服务

最后一步实际上是启动服务。对于很多现代的 Linux 发行版而言，这一步就是 chkconfig \$service on。而其他一些发行版依然需要直接调用初始化脚本。如果传递给脚本的第一个参数是-a，则变量 autostart 会设置为 yes。该变量定义在脚本的开头位置，并且使用比 getopt 更为简单的参数识别方式。它通过检查\$1 是否为-a 来实现。如果\$1 为-a，则 service=\$2 而不是 service=\$1，并且 autostart 会设置为适当的值。使用\$2 表示自动启动会使得脚本更简单，但在语法上与其他 Unix/Linux 脚本相比就会显得不那么自然与一致。

整个多行的 case 语句只有在[autostart == yes]测试成功时才会执行。该测试使用 shell 的 cmd1 && cmd2 结构。其中 cmd1 是实际被测试的条件(第 5 章讨论过，它被链接到)，cmd2 在测试成功时才会运行。如果 autostart 被设置为 yes，则执行后面的 case 语句。否则，

脚本执行 `esac`(终止 `case` 语句)后面的语句。

15.1.5 脚本代码



可从
wrox.com
下载源代码

```
#!/bin/bash

# service is the name of the init script
# as well as the name of the application.
if [ "$1" == "-a" ]; then
    autostart=yes
    service=$2
else
    autostart=no
    service=$1
fi
distro=unknown
init_dir=unknown
rc_dir=/etc/rc.d

# Step 1: Determine the Distribution
if [ -f /etc/redhat-release ]; then
    # Also true for variants of Fedora or RHEL
    distro=redhat
elif [ -f /etc/debian_version ]; then
    # Also true for Ubuntu etc
    distro=debian
elif [ -f /etc/SuSE-brand ] || [ -f /etc/SuSE-release ]; then
    distro=suse
elif [ -f /etc/slackware-version ]; then
    distro=slackware
else
    distro=unknown
fi

# Step 2: Install into the appropriate init directory
case $distro in
    redhat|debian|suse)
        # /etc/rc.d/ is a link to /etc/init.d
        # SuSE and RedHat don't need rc_dir.
        init_dir=/etc/init.d
        rc_dir=/etc
        ;;
    slackware)
        init_dir=/etc/rc.d
        rc_dir=/etc/rc.d
        ;;
    *)
        echo -n "Unknown distribution; guessing init directory... "
        for init_dir in /etc/rc.d/init.d /etc/init.d unknown
        do
```

```

        [ -d ${init_dir} ] && break
    done
    if [ "$init_dir" == "unknown" ]; then
        echo "Failed"
    else
        echo "Found ${init_dir}."
        rc_dir=$init_dir
    fi
esac

if [ $init_dir != unknown ]; then
    cp $service ${init_dir}
else
    echo "Error: Can not determine init.d directory."
    echo "Initialization script has not been copied."
    exit 1
fi

# Step 3: Register the service
case $distro in
    suse|redhat)
        chkconfig --add $service
        ;;
    *)
        ln -sf ${init_dir}/${service} ${rc_dir}/rc2.d/S90$service
        ln -sf ${init_dir}/${service} ${rc_dir}/rc3.d/S90$service
        ln -sf ${init_dir}/${service} ${rc_dir}/rc0.d/K10$service
        ln -sf ${init_dir}/${service} ${rc_dir}/rc6.d/K10$service
        ;;
esac

# Step 4: Start the Service
[ $autostart == yes ] && case $distro in
    suse|redhat)
        chkconfig $service on
        ;;
    unknown)
        echo "Unknown distribution; attempting to start up..."
        ${init_dir}/${service} start
        ;;
    *)
        # Debian, Slackware
        ${init_dir}/${service} start
        ;;
esac
$

```

install-init.sh

15.1.6 调用结果

该脚本很可能被其他脚本调用，例如安装例程。但为了完整性，此处演示了(几乎完全静默的)调用。显示“starting the application!”的是 myapp 初始化脚本，而不是 install-init.sh。


```
# ./install-init.sh -a myapp
/etc/init.d/myapp called with start; starting the application!
# ls -l /etc/init.d/myapp
-rwxr-xr-x 1 root root 429 Apr 11 12:56 /etc/init.d/myapp
# ls -l /etc/rc3.d/S90myapp
-rwxr-xr-x 1 root root 429 Apr 11 12:56 /etc/rc3.d/S90myapp
#
```

15.1.7 小结

不同的发行版随着时间的发展会采取一些稍微不同的方法来处理启动与关闭系统服务。这些方法基本上是相同的，但相比一些比较引人关注的、较为明显的区别，另外一些较小的、微妙的区别则处理起来比较困难。将所有这些复杂度抽象化到单个脚本中意味着系统的其余部分(这里是指应用程序的安装例程)不需要考虑所有这些实现的差异，或者是 shell 脚本如何处理这些差异。同样，对脚本中实现细节的修改可以很方便地独立于主安装程序进行测试。

该脚本使用 shell 中不同的流控制结构来实现脚本中的各个部分。通常而言，在发现 if 被频繁使用后，特别是使用了多个 elif 语句，我们会使用 case 来代替。在这个例子中，if 是识别发行版的理想工具，而 case 是在检测到发行版后执行特定动作的最佳工具。

15.2 实用脚本 15-2: RPM 报告

该脚本所实现的功能是非常有用的，因为它能比较安装在不同机器上的 RPM 包，并在比较结果的基础上生成较为易读的报告。通常而言，RPM 文件名的命名方式使得实现这种功能变得更加困难：短划线可以作为包名称的合法(并且很常见的)部分，也可以用于将包名称、版本号与发行号隔开。

该报告将可以在多个不同机器或者同一机器的不同时刻运行的 rpm -qa 命令的输出作为输入，然后分别比较在不同机器与不同时刻下包列表的区别。对于所有输入文件都相同的 RPM 包，我们用黑色文本将其显示在白色背景下。对于安装了不同版本的包，我们将其显示在灰色背景下以便于识别。

15.2.1 用到的技术

- 参数扩展，特别是%与##
- 关联数组(只能在 4.0 或更高版本的 bash 中使用)
- 函数
- here 文档
- HTML

15.2.2 概念

RPM 文件名中含有大量的信息，但分析起来却不那么容易。例如 gnome-panel-2.16.1-

7.el5，它表示软件的版本号为 2.16.1，发行号是 7.el5。因为文件名的各部分用连字符隔开，且连字符在包名称中又相当常见，所以区分起来比较困难。唯一可行的方法是从文件名的末尾往前进行分析。版本号与发行号不能包含连字符。也就是说，获取版本号与发行号字符串的代码必须去掉包名称与版本号(gnome-panel-2.16.1)，方法是去掉最后一个连字符之后的所有字符，然后去掉剩余字符串中最后一个连字符之前的字符得到 2.16.1。发行号可以很容易地通过去掉最后一个连字符之前的所有字符来得到。稍后将发行号与版本号连起来可以得到一起表示版本号与发行号的单个变量。

在 HTML 中显示简单的数据非常容易，而且不需要深入的 HTML 或 CSS 方面的知识。本脚本生成兼容 HTML 4.01 的 HTML 代码，只使用了<table>元素与一些简单的 CSS。每行以<tr>(表行)开头，并以</tr>结束。行中的每个标题单元都以<th>(表头)开头，并以</th>结束。每个数据单元以<td>(表数据)开头，并以</td>结束。

15.2.3 潜在的陷阱

实现这一功能的主要陷阱在于对包名称中连字符的处理。如果遇到像 dbus-x11-1.1.2-12.el5、xorg-x11-fonts-75dpi-7.1-2.1.el5 和 xorg-x11-drv-vesa-1.3.0-8.1.el5 这样的名称，不借助%、%%、#与##语法似乎不可能从这些名称中得到连贯的数据。此处的关键是对输入格式与处理方法的细心审查。确保 readrpms 函数读取并正确地解析文件名，这样数据就能保持连贯性，而余下的脚本代码也会变得简单。

15.2.4 脚本结构

主要的两个函数是 readrpms 与 showrpms。围绕这两个函数的调用，starhtml 与 endhtml 给出了 HTML 文档起始与结束的基本结构。脚本中的另一个函数是 rpmnames。它只是将文件名中的版本号与发行号去掉，然后对这些 rpm 名称进行排序。showrpms 调用 rpmnames 函数得到排序后的列表，然后创建 HTML 报告。

1. starhtml 与 endhtml

这两个函数显示的是 HTML 输出的开头与结尾的 HTML 代码。starhtml 定义了一个 CSS 样式表并开始定义表格。它使用 here 文档来实现。因为<<-EOF 告诉 here 文档要去掉前面的制表符，所以脚本能具有美观的格式与将函数内容缩进。但是生成的 HTML 是左对齐的，如果编辑 HTML 代码本身可以使其更加美观。

在 starhtml 的末尾以及在 endhtml 中，一个简单的 for 循环在表格的顶部与底部显示标题行，使用的是用于输入的每个文件名。标题可能是主机名、数据收集时刻的时间戳或者是它们两者；也可能是更高级别的信息，如“Web 服务器”与“数据库服务器”。

这两个函数的余下部分只是一些 HTML 文档在开头与结尾所需的 HTML 语法。样式信息定义了 3 种表行：heading、same 与 notsame。它们定义了表示不同种类行的字体大小与背景色。然后，我们通过在脚本中显示<tr class="notsame">来使用这些样式。可以对表格单元格使用不同的颜色。卸载的包可以用一个颜色强调，较早的包用另一种适合需要的颜色。

2. readrpms

`readrpms` 函数读取 `rpm -qa` 输出的文件，并将版本号赋值给一个数组。因为 shell 中没有多维数组，所以不可能有 `rpm[node1][kernel-debug]= 2.6.18-194.el5` 这样的数组。因此应当是用一个名为 `version` 的数组来成对地存储数据，并以 `nodename_rpmname` 作为索引，版本号作为数据。这样就成了 `version[node1_kernel-debug]=2.6.18-194.el5`。相比更加直观的多维数组，它足够为脚本所用。

还存在一些复杂的地方。同一个 RPM 包可能有多个安装版本，所以 `readrpms` 首先使用 `-z` 测试 `node_rpmname` 数组元素是否为空。如果计算机中已有该名称的 RPM 包存在，则将 RPM 包添加到列表。

如 15.2.2 节所讨论的，要获取版本号与发行号，需要使用一个中间变量从包名称的末尾截取相关的数据。

3. showrpms

`showrpms` 函数的作用是显示 HTML 代码。它从 `rpmnames` 得到排序后的包名称列表，然后从第一个节点中为 RPM 获取模板。该函数循环遍历 RPM 包的每个节点。如果它们全部都与模板匹配，则每台机器安装的该 RPM 包都具有相同的版本号；否则进行标记，且相应的行可以用适当的 CSS 标签来标记。循环中的 `break` 具有过早优化的嫌疑，但如果要比较大量的系统，则在发现存在差异时不再继续检查这样的做法可能获得一些性能上的改进。为了让输出保持简洁且可供搜索，在 RPM 包都相同的情况下显示 `RPM MATCH`，只有在不同的情况下显示每个节点。

随后使用 `for` 循环再次遍历每个值，并简单地向 HTML 文件回显 `<td>${version[$idx]}</td>`。两个因素使得这一操作变得复杂。第一，如果某个包没有安装，相比于将表格单元空格空着，不如清楚地说明没有安装。第二，如果安装了 RPM 包的多个副本，则将每个副本分别放在每一行会显得比较美观，所以需要在 `readrpms` 函数添加空格的地方插入一个 `
` 标签。`readrpms` 本身可以插入 `
` 标签，但最好将数据结构与输出格式分开。这样一来，即使最后的输出结果是纯文本文件、RTF、CSV 或者一些其他适当的格式，我们还是可以使用同一段代码。

因为该脚本向 `stdout` 与 HTML 文件显示输出，所以只有在 RPM 包不都相同的时候，输出的第一行才流向 `stdout`。接下来的一行写入 HTML 文件。如果数组元素为空，`${version[$idx]:-NotInstalled}` 显示版本号或者 `NotInstalled`。`sed` 将版本号(表示多个版本)中的任何空格替换为 `
` 标签。最后，`NotInstalled` 标志被转换回 `Not Installed`。

15.2.5 脚本代码



可从
wrox.com
下载源代码

```
#!/bin/bash
```

```
declare -A version
```

```
HTML=report.html
```

```
function rpmnames
{
```

```

for rpm in `cat $* | sort -u`
do
    echo ${rpm%-*-*}
done | sort -u
}

function readrpms
{
    for node in $*
    do
        while read rpm
        do
            # rpm is gnome-panel-2.16.1-7.el5
            rpmname=${rpm%-*-*}           # gnome-panel
            rpmnameversion=${rpm%-*}      # gnome-panel-2.16.1
            rpmversion=${rpmnameversion##*-} # 2.16.1
            rpmrelease=${rpm##*-}         # 7.el5
            idx=${node}_${rpmname}
            if [ -z "${version[$idx]}" ]; then
                version[$idx]="${rpmversion}-${rpmrelease}"
            else
                version[$idx]="${version[$idx]} ${rpmversion}-${rpmrelease}"
            fi
        done < $node
    done
}

function showrpms
{
    for rpmname in `rpmnames $*`
    do
        idx=$1_${rpmname}
        template="${version[$idx]}"
        allsame=1
        for node in $*
        do
            idx=${node}_${rpmname}
            if [ "${version[$idx]}" != "${template}" ]; then
                allsame=0
                break
            fi
        done

        if [ $allsame -eq 1 ]; then
            echo "RPM MATCH: $rpmname $template"
            echo "<tr class=\"same\"> >> $HTML"
        else
            echo "RPM $rpmname"
            echo "<tr class=\"notsame\"> >> $HTML"
        fi
    done
}

```

```
echo "<th>${rpmname}</th>" >> $HTML
for node in $*
do
    idx=${node}_${rpmname}
    [ $allsame -eq 0 ] && echo "$node : ${version[$idx]:-Not Installed}"
    echo "<td>${version[$idx]:-NotInstalled}</td>" | \
        sed s/" "  
"/>"/g | \
        sed s/"NotInstalled"/"Not Installed"/g >> $HTML
done
echo "</tr>" >> $HTML
done
}

function starthtml
{
    cat - <<-EOF > $HTML
        <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
            "http://www.w3.org/TR/html4/loose.dtd">
        <html>
        <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" >
        <title>Report on $*</title>
        <style type="text/css">
            tr.heading { background-color: #f2f2f2; font-size: 1.2em; }
            tr.same { background-color: white; }
            tr.otsame { background-color: #c2c2c2; }
            td { font-family: sans-serif; font-size: 0.8em; }
            th { font-family: serif; font-size: 0.8em; }
        </style>
        </head>
        <body>
        <table border="1">
        <tr class="heading"><th>RPM</th>
EOF

        for node in $*
        do
            echo "<th>$node</th>" >> $HTML
        done
        echo "</tr>" >> $HTML
    }

function endhtml
{
    echo "<tr class=\"heading\"><th>RPM</th>" >> $HTML
    for node in $*
    do
        echo "<th>$node</th>" >> $HTML
    done
    echo "</tr></table>" >> $HTML
}
```

```

    echo "</body></html>" >> $HTML
}

starthtml $*
readrpms $*
showrpms $*
endhtml $*

```

rpm-report.sh

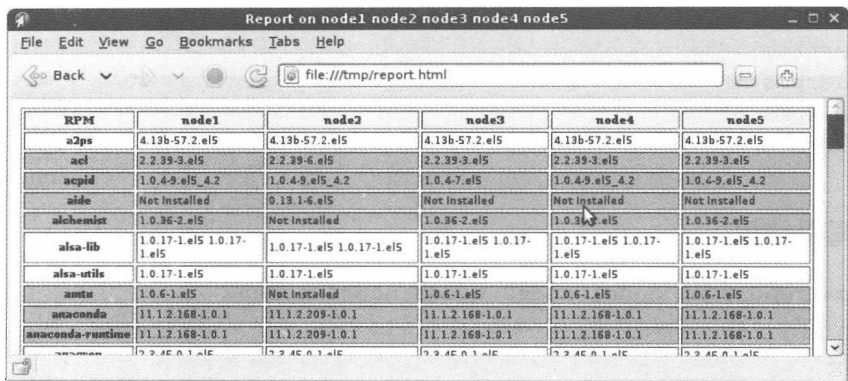
15.2.6 调用结果

```

$ ./rpm-report.sh node1 node2 node3 node4 node5
RPM MATCH: a2ps 4.13b-57.2.el5
RPM acl
node1 : 2.2.39-3.el5
node2 : 2.2.39-6.el5
node3 : 2.2.39-3.el5
node4 : 2.2.39-3.el5
node5 : 2.2.39-3.el5
RPM acpid
node1 : 1.0.4-9.el5_4.2
node2 : 1.0.4-9.el5_4.2
node3 : 1.0.4-7.el5
node4 : 1.0.4-9.el5_4.2
node5 : 1.0.4-9.el5_4.2
RPM aide
node1 : Not Installed
node2 : 0.13.1-6.el5
node3 : Not Installed
node4 : Not Installed
node5 : Not Installed
RPM alchemist
node1 : 1.0.36-2.el5
node2 : Not Installed
node3 : 1.0.36-2.el5
node4 : 1.0.36-2.el5
node5 : 1.0.36-2.el5
RPM MATCH: alsa-lib 1.0.17-1.el5 1.0.17-1.el5
RPM MATCH: alsa-utils 1.0.17-1.el5
. . . . .
RPM zenity
node1 : 2.16.0-2.el5
node2 : Not Installed
node3 : 2.16.0-2.el5
node4 : 2.16.0-2.el5
node5 : 2.16.0-2.el5
RPM MATCH: zip 2.31-2.el5
RPM MATCH: zlib 1.2.3-3 1.2.3-3
$ web-browser ./report.html

```

调用结果如图 15-1 所示。



RPM	node1	node2	node3	node4	node5
a2ps	4.13b-57.2.el5	4.13b-57.2.el5	4.13b-57.2.el5	4.13b-57.2.el5	4.13b-57.2.el5
ac1	2.2.39-3.el5	2.2.39-6.el5	2.2.39-3.el5	2.2.39-3.el5	2.2.39-3.el5
acpid	1.0.4-9.el5_4.2	1.0.4-9.el5_4.2	1.0.4-7.el5	1.0.4-9.el5_4.2	1.0.4-9.el5_4.2
aide	Not Installed	8.13.1-6.el5	Not Installed	Not Installed	Not Installed
alchemy	1.0.36-2.el5	Not Installed	1.0.36-2.el5	1.0.36-2.el5	1.0.36-2.el5
alsa-lib	1.0.17-1.el5 1.0.17-1.el5	1.0.17-1.el5 1.0.17-1.el5	1.0.17-1.el5 1.0.17-1.el5	1.0.17-1.el5 1.0.17-1.el5	1.0.17-1.el5 1.0.17-1.el5
alsa-utils	1.0.17-1.el5	1.0.17-1.el5	1.0.17-1.el5	1.0.17-1.el5	1.0.17-1.el5
amtu	1.0.6-1.el5	Not Installed	1.0.6-1.el5	1.0.6-1.el5	1.0.6-1.el5
anaconda	11.1.2.168-1.0.1	11.1.2.209-1.0.1	11.1.2.168-1.0.1	11.1.2.168-1.0.1	11.1.2.168-1.0.1
anaconda-runtime	11.1.2.168-1.0.1	11.1.2.209-1.0.1	11.1.2.168-1.0.1	11.1.2.168-1.0.1	11.1.2.168-1.0.1

图 15-1

15.2.7 小结

该脚本生成的报告很有用，且脚本本身简短而高效，这是缘于使用了各种不同的变量结构，尤其是数组与参数扩展。将报告的内容与一些很简单的 HTML 组合起来，我们可以得到一份看起来比较专业的报告，并且可以将它粘贴到文档、放到网络上，甚至是打印出来(尽管这里用到的测试数据会横向打印成 42 页)。

代码的结构中没有对可以比较的主机或 RPM 包数目进行限制。这与很多执行类似任务的脚本不同，后者通常会对\$node1、\$node2 与\$node3 的值进行硬编码，因此不能处理更多的数目。输出的 HTML 可能很难一次全部看到，但脚本中从 RPM 开始显示的输出可以很容易通过相关信息被其他脚本搜索到。

将一些非常简单的技术按照正确的方式组合起来，由此得到的脚本与所要完成的任务能很好地契合。如果使用一种略微不同格式的数据，则可能不会那么简单。但为了清晰与方便进一步扩展，还是可以将其分解为独立的读取、显示与格式化 3 个部分，这样就相当简单了。

15.3 实用脚本 15-3: postinstall 脚本

使用 Kickstart 进行自动安装是一种很有用的向很多类似机器程序化安装软件的方法。每台机器的不同之处在于它们的 IP 地址。Kickstart 可以在安装过程中配置 IP 地址，但不能配置额外的网络或网口绑定。该脚本可以适配 Kickstart 文件的%post 部分来对网络进行配置，然后在下次启动的时候使用新的网络参数。

15.3.1 用到的技术

- Kickstart
- Red Hat Enterprise Linux 下的网络配置
- 网口绑定

- 函数
- here 文档
- ping

15.3.2 概念

联网是计算机配置的关键部分。如果系统在网络中不可见，则系统其余部分的工作再出色也没什么用处——它无法向外界提供服务。网口绑定可以防止某些常见的故障导致系统完全掉线。如果为网络流量使用两个网络适配器，当其中一个在路由中失效后，则可以使用另一个。最好使用两种不同类型的网络适配器，使得它们具有不同的实现与不同的内核驱动程序。当然还需要两根不同的电缆线，并且要接到独立的网络交换机，这样系统可以容忍某个设备出现故障。

Linux 有很多不同的绑定模式，各自有不同的特性。绑定模块加载到内核中的时候会指定绑定模式。

- 模式 0 或者 **balance-rr**：循环方式，目的是平衡负载与容错。
- 模式 1 或者 **active-backup**：用于容错的故障恢复模式。本节介绍的脚本就是配置成了该模式。两个网络适配器(其中一个在任何时刻都是活动状态)以及另外一个称为 **bond0** 的(虚拟)设备都用 IP 地址配置。
- 模式 2 或者 **balance-xor**：使用源与目标 MAC 地址的异或结果来决定每个外发包使用哪个网络适配器。
- 模式 3 或者 **broadcast**：一次在所有的接口上进行传输，用于容错。
- 模式 4 或者 **802.3ad**：链路聚合，用于容错与负载平衡。
- 模式 5 或者 **balance-tlb**：从最不繁忙的网络适配器发送网络包，用于负载平衡。
- 模式 6 或者 **balance-alb**：对传入包也进行负载平衡，方法是调整远程 ARP 缓存使其向一个或另一个适配器发送。

尽管 <http://www.kernel.org/doc/Documentation/networking/bonding.txt> 与 `/usr/share/doc/kernel-doc-*/Documentation/networking/bonding.txt` 中的内核文档简洁地提供了大量关于网口绑定的有用信息，但 Internet 上也有很多关于它的介绍。本脚本设置的选项为 `miimon=100`，它让网口绑定每 100 毫秒检查一次链接(每秒 10 次)。`fail_over_mac=1` 让网络适配器保留自己的 MAC 地址，故障恢复后更新远程 ARP 缓存。很多这样的选项在 `bonding.txt` 文件中都有相关文档说明。`bonding.txt` 文件可以在上面提到的位置找到。

系统运行时可以通过 `cat /proc/net/bonding/bond0` 对网口绑定状态进行查看。该条命令提供了关于绑定状态本身与绑定的子设备相当细节的信息。

15.3.3 潜在的陷阱

自动网络配置最坏的结果之一是计算机最终无法通过网络访问。更糟糕的情况是，计算机无法访问的原因在于它使用的 IP 地址有其他机器正在使用。将新的机器连接到网络需要一些合法性检测，所以该功能库在对设备进行配置之前进行一些基本的检查。

网络设备命名不是总可以完全预测的。有各种各样的方法对 `udev` 子系统进行配置，以使一些特殊端口总是被称为 `eth0`、`eth1` 或任意所需名称。

15.3.4 脚本结构

Kickstart 文件定义了一个被 `bash` 执行的 `%post` 部分。该部分可以用来按照任意所需方式调整系统配置。如果我们在安装服务器中提供一个简单的库，则不需要在每个服务器的 Kickstart 文件的 `%post` 部分进行复杂的脚本编程。

该库提供了 3 个函数。第一个在 `addroute` 文件中，作用是向设备添加一个静态的网络路由。在 Red Hat 中是通过 `/etc/sysconfig/network-scripts` 中的 `route-NIC` 文件来完成的。每个网络适配器都可以有一个这样的文件，并且路由被添加到特定的适配器。尽管在安装阶段我们可以安全地假设该文件已经存在，不过使用 `>>` 追加比使用 `>` 简单地写入要安全，这样已有的路由也不会丢失。

该文件中的第二个函数 `makeslave` 是为了使第三个函数能简单一些。该函数使用了 `here` 文档为要被设置成虚拟绑定设备的子设备的网络适配器创建配置文件。它接收两个参数：要创建的网络适配器与所属的网口绑定。这时最好使用单个 `>` 来确保任何已有的适配器配置都被删除。



因为只在库内部调用，所以假设 `makeslave` 已经从 `/etc/sysconfig/network-scripts` 目录中运行比较安全。向其他脚本或库提供接口的脚本不应当作这样的假设。

第三个函数 `addbond` 是库的主要部分。该函数使用动态 IP 地址配置虚拟绑定设备。函数运行时，动态 IP 地址会被用到最为合适的 NIC，在检测到故障时会从一个适配器浮动到另一个适配器。

函数的开头读取传递给它的变量，然后检查提供的参数是否齐全，并将其作为一个基本的合理性检查。如果 `$5` 的长度不是 0，则它之前的其他参数可以认为有值。该脚本并不面向用户，并且没有不合理地认为提供的参数都是合法的。脚本随后再执行一次基本的测试。因为网络已经在系统中经过了配置(虽然只有一个网络适配器)，所以应当可以肯定的是，被安装的机器所赋予的 IP 地址不会已经被分配给网络中的任何其他主机。但有一个例外——绑定的 IP 地址已经被作为执行安装的 IP 地址，并且这样的情况时有发生。因此该脚本检测 `ip` 命令是否在其输出中列出 `inet $IP`。如果被安装的机器还没有使用绑定的 IP 地址，使用 `ping` 命令可以确认还没有被网络中的其他部分使用。这个简单的测试确保机器上线时使用的 IP 地址不会已经被网络上的其他机器使用。

因为无法改变安装环境，且可能包含各种预料不到的配置，所以需要更多的合理性检查以确保绑定本身没有经过配置。如果 `/etc/sysconfig/network-scripts` 中有 `ifcfg-$BOND` 文件，或者 `/etc/modprobe.conf` 中有一个相应的绑定项，则该脚本也会提前退出，因为需要手动调整。

创建绑定需要两个子设备。上面提到的 `makeslave` 函数使用两个子设备创建绑定，并且避免了 `addbond` 函数中的重复。`makeslave` 中使用了一个非常类似的 `here` 文档创建绑定设备项。`here` 文档会向配置添加 IP 地址与网络掩码。绑定项还会添加到 `/etc/modprobe.conf`

中。这样做的目的是在内核启动时加载绑定模块，以及为模块设置相应的选项。

最后，已有的子接口被关闭，然后加载绑定模块，新配置的绑定设备上线。严格来说，最后这一步不是必要的，因为系统已经能够与 Kickstart 服务器进行通信，并且新的设备会在第一次重启时上线。重启是在 Kickstart 文件的%post 部分完成之后立即执行。

client-ks.cfg 文件只显示 Kickstart 文件的%post 部分。该部分配置 bond0 使用 eth0 与 eth4(在有 4 个网络端口的系统中，eth0 与 eth4 应当是系统中主板上的第一个端口与 PCI 网卡的第一个端口)，然后配置 bond1 使用 eth1 与 eth5。eth1 与 eth5 应当是前面两个端口的邻近端口。

15.3.5 脚本代码



可从
wrox.com
下载源代码

```
# Library of networking functions
# Assumes RedHat Enterprise Linux style
[ -f /etc/redhat-release ] || return

function addroute
{
    # Add a route to a device
    # $1 = network adapter (eg eth0, bond0)
    # $2 = destination
    # $3 = router
    cd /etc/sysconfig/network-scripts
    echo "Adding $2 via $3 on $1"

    echo "$2 via $3" >> route-$1
}

function makeslave
{
    # $1 = network adapter
    # $2 = bond
    cat - > ifcfg-$1 <<EOF
DEVICE=$1
BOOTPROTO=none
ONBOOT=yes
MASTER=$2
SLAVE=yes
USERCTL=no
EOF
}

function addbond
{
    # $1 = bond, $2=network adapter 1, $3 = network adapter 2
    # $4 = IP address or name, $5 = netmask
    BOND=$1
    DEV1=$2
    DEV2=$3
```

```
IP=`getent hosts $4 | awk '{ print $1 }'`
NAME=`getent hosts $4 | awk '{ print $1 }'`
NETMASK=$5

if [ -z "$NAME" ] || [ -z "$5" ]; then
    echo "Usage: addbond bond dev1 dev2 ip netmask"
    return 1
fi

/bin/ip a | grep "^    inet ${IP}/" > /dev/null
if [ "$?" -ne "0" ]; then
    if ping -c1 -w1 $IP > /dev/null 2>&1
    then
        echo "Error: $NAME ($IP) is responding to ping. Not configuring $IP"
        return
    fi
fi

cd /etc/sysconfig/network-scripts
if [ -f ifcfg-$BOND ]; then
    echo "Error: $BOND is already configured"
fi
[ -f ifcfg-$DEV1 ] && mv ifcfg-$DEV1 bak.ifcfg-$DEV1
[ -f ifcfg-$DEV2 ] && mv ifcfg-$DEV2 bak.ifcfg-$DEV2

if grep $BOND /etc/modprobe.conf > /dev/null
then
    echo "Error: $BOND is already defined in /etc/modprobe.conf"
    return
fi

echo "Creating bond device $BOND from $DEV1 and $DEV2"
echo "with the IP address ${IP}/${NETMASK}"

makeslave $DEV1 $BOND
makeslave $DEV2 $BOND

cat - > ifcfg-$BOND <<EOF
DEVICE=$BOND
BOOTPROTO=none
IPADDR=$IP
NETMASK=$NETMASK
ONBOOT=yes
EOF

cat - >> /etc/modprobe.conf << EOF
alias $BOND bonding
options $BOND mode=1 miimon=100 fail_over_mac=1
EOF
```

```

    ifdown $DEV1
    ifdown $DEV2
    modprobe $BOND
    ifup $BOND
}

```

netlib

```

%post
. /mnt/source/netlib

```

```

makebond bond0 eth0 eth4 192.168.1.53 255.255.255.0
addroute bond0 192.168.9.0 192.168.1.1

```

```

makebond bond1 eth1 eth4 192.168.2.53 255.255.255.0

```

client-ks.cfg

15.3.6 调用结果

```

Creating bond device bond0 from eth0 and eth4
with the IP address 192.168.1.53/255.255.255.0
Adding 192.168.9.0 via 192.168.1.1 on bond0
Creating bond device bond1 from eth1 and eth4
with the IP address 192.168.2.53/255.255.255.0

```

15.3.7 小结

安装过程中的环境非常小，但在安装的 `postinstall` 阶段可以进行很多自定义。网口绑定可以在该阶段进行配置，并且使用代码库来实现是不错的方式，因为使用库可以保持 Kickstart 文件本身的简洁。另外，对脚本的任何修改只需要应用于 Kickstart 服务器上的副本，而不需要应用到成百(甚至成千)的客户端 Kickstart 文件。

第 16 章

系 统 管 理

本章为日常的系统管理任务提供了 4 个实用脚本。第 1 个实用脚本是系统启动时自动启动应用程序的初始化脚本。其中展示了 `case` 的使用以及利用 `/var/run` 文件系统存储 PID 文件。

第 2 个实用脚本提供了两个相关的 CGI 脚本，它们分别处理 GET 与 POST 请求。该例子演示了如何处理这两种从浏览器向服务器传递数据的方法，还介绍了用户提交数据处理中的安全问题。

第 3 个实用脚本展示了如何使用配置文件提供默认值并向最终用户显示这些值，以及记住用户先前选择的值。

最后第 4 个实用脚本实现了一个锁机制，用来确保多个并发进程可以共享一个临界资源，而又不会干扰其他进程的使用。

16.1 实用脚本 16-1：初始化脚本

系统启动脚本经常被称为初始化脚本，因为它们是由 `/etc/init.d` 目录中的 `init` 守护进程来启动的。本脚本为这样的初始化脚本提供了一个基本结构。进程在后台启动，并且一直以守护进程的形式运行。同一个进程被关闭，用以停止守护进程。这并不适用于所有应用程序。例如，可能 `$INSTDIR/$APP` 启动了另外 3 个进程，然后立刻退出。Apache Web 服务器的 `apachectl` 命令就是这样一个典范。在这种情况下，需要对应用程序的细节有更多的了解。如何按照需求监控或停止这些子进程呢？然而，对于守护进程这样的系统，本脚本是一个很好的入门脚本。



后台进程通常被称为守护进程。

16.1.1 用到的技术

- init
- chkconfig
- case

16.1.2 概念

最近的一些操作系统，尤其是 Ubuntu 与 Solaris，为了简单地实现平行系统，开始背离传统的初始化脚本。它们使用的工具分别是 Upstart 与 SMF。另外一种 Unix 即 BSD 也已经使用了一种不太相同的初始化脚本的变种。同样地，尽管 Slackware 现在也支持大多数当前的 Linux 发行版所青睐的 SysV 初始化脚本类型，但它使用的初始化脚本更类似于 BSD。

初始化脚本负责应用程序的启动与关闭，以及对特定应用程序状态的监控与维护。初始化脚本的两个标准选项是 start 与 stop。当调用 chkconfig 来检查所有已安装的初始化脚本的状态时，调用使用 status 选项。非常简单的第三方启动脚本会默认启动其服务，除非调用时使用了 stop 选项。这些脚本在每次进行系统状态测试时会启动自身的一个副本。

Linux 标准规范(Linux Standard Base, LSB)还要求实现 restart、force-reload 与 status 这些选项。LSB 还建议实现 try-restart(只有在服务已经运行的情况下才会进行重启)与 reload(不关闭或启动服务就对配置进行重新读取)。我们可以在线阅读完整的规范说明；当前的 4.1.0 规范在 http://refspecs.freestandards.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/tocsysinit.html 中可以找到，并不需要完全按照规范进行，但本章后面 16.1.3 节介绍的 lsb-ourdb 模板对大多数应用程序都适用。对 Provides 进行简单编辑：匹配服务名称的文本行，修改两个 Description 行用以显示适当的信息。



服务的安装与注册在第 15 章的实用脚本 15-1 的“注册服务”一节有介绍。

1. 启动

当使用 start 选项调用脚本时，脚本检测已经在运行的进程实例。如果找到 PID 文件，则关闭相关进程。否则，脚本在后台启动应用程序，记录其 PID(在\$!中)，然后存储在 \$PIDFILE 中。

2. 关闭

当使用 stop 选项调用脚本时，脚本检测应用程序的状态。如果应用程序没有在运行，则 status 会以非零返回码退出。所以 \$0 status || exit 1 表示，如果 status 没有找到正在运行的副本，则该运行实例退出。因为脚本已通过 status 调用确认了应用程序正在运行，所以可以从 \$PIDFILE 中获取 PID，并将其关闭。如果 kill 成功，则脚本删除 \$PIDFILE 文件；否则以返回码 1 退出。

3. 状态

要检查状态，脚本必须首先检查\$PIDFILE，然后使用 `ps -o comm= -p $PID` 来获取用该PID运行的进程的名称。如果名称与\$APP不同，则返回非零退出码表示失败。否则，脚本调用 `ps -p $PID` 并返回 `ps` 返回的退出码(0表示成功，非零表示失败)。



`ps -o comm` 显示标题与实际进程名称。`ps -o comm=`不显示标题。

4. 重启与强制重载

如果要重启，脚本只需要使用 `stop` 参数调用自己。如果成功，则再次使用 `start` 参数调用自己。`force-reload`的实现也是LSB要求的，但在这里，它与 `restart` 相同，所以调用\$0 `restart`。关于 `force-reload`的更加特定的实现在以后需要的时候再进行添加。

5. 默认参数

因为任何时候都可能增加新的参数，所以能对一切进行匹配的*选项会向标准输出显示一条消息，然后用2作为返回码退出。返回码2通常用来表示用法错误。注意，如果没有传递任何参数也会执行这一段代码块。

16.1.3 潜在的陷阱

不同的实现使用的注册服务的方法各不相同。LSB特别对格式化的注释进行了定义。这些注释可以被 `chkconfig` 用来进行服务注册。LSB标准建议的格式如下：

```
### BEGIN INIT INFO
# Provides: lsb-ourdb
# Required-Start: $local_fs $network $remote_fs
# Required-Stop: $local_fs $network $remote_fs
# Default-Start: 2 3 4 5
# Default-Stop: 0 1 6
# Short-Description: start and stop OurDB
# Description: OurDB is a very fast and reliable database
#                engine used for illustrating init scripts
### END INIT INFO
```

在实际情况下，初始化脚本可能包含这些字段的一部分或全部，并且也可能包含一些隐藏在注释中的其他配置数据。例如，在Red Hat Enterprise Linux 6中，`/etc/init.d/sshd`的开头部分就有一些额外的特殊注释字段。Red Hat也在头部注释中包含了一些其他信息，但### BEGIN INIT INFO与### END INIT INFO之间的注释要求用来对脚本进行注册。

```
# sshd                Start up the OpenSSH server daemon
#
# chkconfig: 2345 55 25
# description: SSH is a protocol for secure remote shell access. \
```



```
#           This service starts up the OpenSSH server daemon.
#
# processname: sshd
# config: /etc/ssh/ssh_host_key
# config: /etc/ssh/ssh_host_key.pub
# config: /etc/ssh/ssh_random_seed
# config: /etc/ssh/sshd_config
# pidfile: /var/run/sshd.pid

### BEGIN INIT INFO
# Provides: sshd
# Required-Start: $local_fs $network $syslog
# Required-Stop: $local_fs $syslog
# Should-Start: $syslog
# Should-Stop: $network $syslog
# Default-Start: 2 3 4 5
# Default-Stop: 0 1 6
# Short-Description: Start up the OpenSSH server daemon
# Description:      SSH is a protocol for secure remote shell access.
#                   This service starts up the OpenSSH server daemon.
### END INIT INFO
```

16.1.4 脚本结构

初始化脚本以其最简单的形式包含了一个对第一个参数进行的测试, 并使用 `start` 或 `stop` 参数表示启动或关闭应用程序。如前面介绍的, 也应当实现 `status`、`restart` 与 `force-reload` 以保持对 LSB 的最小兼容性。

1. chkconfig

对于 `chkconfig` 的使用, 脚本从一个头部开始。该头部被格式化为 `shell` 脚本注释的形式。这些注释在当成脚本处理的时候会被忽略, 但安装与描述服务的时候会被 `chkconfig` 实用程序读取。在没有 `chkconfig` 的系统中, 这些注释仅被当成注释, 而且会被忽略。

2. start 与 stop

在头部之后, 代码的主体通常用 `case` 语句的形式实现。`case` 语句会根据复杂度用函数或在 `case` 语句内部运行正确的命令。LSB 标准要求初始化脚本应当根据结果返回不同的返回码。很多服务无论处于什么状态一般都返回 0, 而其他的只要出错就返回 1 或 -1。

3. Provides 字段与 Required-Start 字段

头部注释中有一个名为 `Provides` 的字段, 它定义了该初始化脚本提供的服务器的名称。还有名为 `Required-Start` 与 `Required-Stop` 的字段, 它们与 `Provides` 字段组合起来用于决定脚本的运行顺序。如果 `/etc/init.d/udev` 声明了 `Provides: udev`, 且 `/etc/init.d/network-manager` 声明了 `Required-Start: $remote_fs dbus udev`, 则 `/etc/init.d/udev` 将在 `/etc/init.d/network-manager` 之前运行。

另一个比较特殊的字段是\$remote_fs。它是一个系统定义的设施，在名称的开头用美元符号进行标记。当前有 7 个系统定义的设施，如表 16-1 所示。

表 16-1 系统设施

名 称	描 述
\$local_fs	所有本地文件系统已成功挂载
\$network	网络子系统可用
\$named	IP/主机名查找(如 DNS)可用
\$portmap	RPC 服务可用
\$remote_fs	远程(网络)文件系统可用
\$syslog	系统日志器可用
\$time	系统时钟准确

16.1.5 脚本代码

```
#!/bin/bash

### BEGIN INIT INFO
# Provides: myapp
# Required-Start: $local_fs $network $remote_fs
# Required-Stop: $local_fs $network $remote_fs
# Default-Start: 2 3 4 5
# Default-Stop: 0 1 6
# Short-Description: start and stop myapp
# Description: MyApplication is a great utility for
#              doing things with systems.
### END INIT INFO

INSTDIR=/usr/local/bin
PIDFILE=/var/run/myapp.pid
APP=myapp

case $1 in
  start)
    if [ -f $PIDFILE ]; then
      echo "Error: $PIDFILE already exists."
      exit 1
    fi
    $INSTDIR/$APP &
    PID=$!
    echo $PID > $PIDFILE
    exit 0
  ;;
```

```

stop)
    $0 status || exit 1
    PID=`cat $PIDFILE 2>/dev/null`
    if [ "$?" -eq "0" ]; then
        kill -9 $PID && rm -f $PIDFILE || exit 1
    else
        exit 1
    fi
    exit 0
;;

status)
    PID=`cat $PIDFILE 2>/dev/null`
    if [ "$?" -ne "0" ]; then
        exit 1
    fi
    if [ -f $PIDFILE ]; then
        if [ "`ps -o comm= -p $PID`" != "$APP" ]; then
            echo "Error: PID $PID is not $APP"
            exit 1
        fi
        ps -p $PID > /dev/null 2>&1
        exit $?
    else
        exit 1
    fi
    ;;

restart)
    $0 stop && $0 start
    ;;

force-reload)
    $0 restart
    ;;

*) echo "Argument \"$1\" not implemented."
   exit 2
   ;;

esac

```

16.1.6 调用结果

```

# /etc/init.d/myapp status ←—— 如果没有在运行, 则 status 参数返回非零值。
# echo $?
1
# /etc/init.d/myapp start ←—— 启动应用程序使 PID 写入 myapp.pid 文件中。
# cat /var/run/myapp.pid
9024
# ps -fp `cat /var/run/myapp.pid`
UID  PID PPID  C  STIME TTY          TIME CMD
Root 9024    1   0 12:04 pts/3      00:00:00 /bin/bash /etc/init.d/myapp star

```

```
# /etc/init.d/myapp restart ← 重启应用程序会有新的 PID，因为旧的实例已关闭，
# cat /var/run/myapp.pid      且新的实例在 12:09 时刻启动。
9040
# ps -fp `cat /var/run/myapp.pid`
UID    PID  PPID  C  STIME TTY          TIME CMD
Root 9040    1   0 12:09 pts/3    00:00:00 /bin/bash /etc/init.d/myapp star
# /etc/init.d/myapp stop ← 关闭应用程序会导致 myapp.pid 文件的删除。
# cat /var/run/myapp.pid
cat: /var/run/myapp.pid: No such file or directory
# /etc/init.d/myapp restart ← 如果应用程序没有运行，则 restart 不会将其启动。
# cat /var/run/myapp.pid
cat: /var/run/myapp.pid: No such file or directory
```

16.1.7 小结

初始化脚本一般非常简单，但根据系统的不同，它们的实现也不相同。该脚本提供了一个足够简单的对大多数配置都有效的框架。但如果在脚本开头的注释中提供一些额外的信息，则在并行启动与依赖关系检测方面，一些系统可以做到更加灵活。

16.2 实用脚本 16-2: CGI 脚本

通用网关接口(Common Gateway Interface, CGI)是定义数据如何传输到 Web 服务器的协议。协议的大部分都组织到 <http://www.example.com/page?name=steve&shell=bash> 这个网站上。但还有不是很引人注目的是 Web 服务器处理表单的方式。表单通常由 PHP 这样的语言处理。这就需要在 Web 服务器之上添加额外的软件，但仅仅使用 Web 服务器与 shell 就可以完成这些任务。

在如今的 Internet 上，CGI 脚本需要极强的健壮性与安全性，因为任何可能欺骗脚本使其执行其他操作的人都可能在运行脚本的用户账户的权限下执行任何代码。像 PHP 这样更复杂的系统会使系统体积膨胀，并能将发生的细节隐藏起来，但这些系统确实增加了额外的安全保护措施。对于这些更复杂系统或者受信任或非常简单环境下的调试问题，shell 同样可以用于处理 CGI 脚本。

16.2.1 用到的技术

- HTTP
- CGI; RFC 3875
- Apache mod_cgi
- eval、case、read

16.2.2 概念

CGI 协议的演进比其文档化的过程要快，但是 RFC 3875(<http://www.ietf.org/rfc/rfc3875>)已经完成了对通用网关接口的文档化。CGI 允许 Web 服务器使用 GET 与 POST 这两个协

议从浏览器接收额外的数据。这两个协议都由 HTTP 协议定义。尽管 DELETE 与 PUT 没有用在 Web 技术中，但在表述性状态转移(Representational State Transfer, REST)架构中也能找到它们。

1. GET

最简单的表单是 GET，它在 URL 中嵌入参数。这作为环境变量 QUERY_STRING 被传递给脚本。由脚本来分析该变量是否最适合，但标准的形式是每个 variable=value 对在初始的问号之后发送，然后&符号将变量与值分开。这就是在用表单发送 GET 请求时变量的表示方法。另外还进行了一些小小的编码。URI 只能包含某些字符，所以空格用%20(空格的 ASCII 表示是 0x20)来代替，尽管加号也可以用来表示空格。分号变成%3A、斜杠变成%2F 等。

2. POST

另一个常用表单是 POST，它在请求的主体中传递参数。我们可以使用 POST 从浏览器向 Web 服务器发送文件。POST 没有使用 QUERY_STRING 变量，而是在脚本的标准输出中处理数据。

3. 表单

发送数据的表单被定义成 HTML。表单具有定义好的动作(action)与方法(method)。HTML 本身不必具备 CGI 的功能。它可以是纯 HTML 网页。CGI 脚本的 URL 定义在 action 参数中，而 method 指定浏览器使用 GET 或 POST 向 Web 服务器发送数据。

16.2.3 潜在的陷阱

在处理用户提交的数据时主要的陷阱就在于安全。例如，在测试本地或整个受信系统时，如果我们可以避免这一风险，则使用 shell 作为调试工具可以非常节省时间，因为它能避免复杂的第三方软件带来的系统体积膨胀。

如下面的 GET 一节所介绍的，编辑地址栏可用来从 Web 浏览器提取潜在的敏感数据，这在 16.2.6 节的图 16-5 中也有显示。用户也可以将 telnet 会话导向 Web 服务器的 80 端口，然后发送任何直接指向 CGI 脚本的数据。发送方式可以是 QUERY_STRING 变量(对于 GET 请求)或者标准输入(对于 POST 请求)。管理员很难防范所有可能的攻击，时间越长越是如此，因为到目前为止 CGI 脚本已经收到过多种形式的攻击。在理想情况下，CGI 脚本会准确地知道它所期望获取的输入并丢弃任何其他数据而不进行任何处理。在实际情况下，这通常是不可能的，因为必须认为所有的数据都具有潜在的恶意代码。要检查的内容包括：

- \$VARIABLE
- \${VARIABLE}
- cmd
- `cmd`
- \$(cmd)
- cmd > file (或>>)

- `cmd < file` (或`<<`、`<<-`、`<<<`)

16.2.4 脚本结构

首先, 假设我们正在使用 Apache Web 服务器, 将类似下面的代码添加到 Apache 配置中来开启 CGI 功能。根据具体的设置情况, 文件名与目录各不相同, 但重要的是定义 `/cgi-bin` 别名, 以及为别名目录设置 `+ExecCGI` 选项。然后将 `index.html` 文件安装到 Web 服务器的文档目录下, 以及将两个 CGI 文件安装到 `cgi-bin` 目录下。

```
ScriptAlias /cgi-bin/ /var/www/cgi-bin/
<Directory "/var/www/cgi-bin">
    AllowOverride None
    Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
    Order allow,deny
    Allow from all
</Directory>
```



为了使用 `example.com` 域名获得一些截屏, 我们在 `/etc/hosts` 中进行了一项定义, 并在 Apache 中创建一个虚拟主机来响应该域名。我们在该脚本后面的 POST 一节可以看到 `/etc/hosts` 输出中的这一域名。

1. 头部

脚本必须只能产生 HTML 文档。所有这些 HTML 文档都必须发送到浏览器。发送内容必须以一个两行的头部开始。第一行定义发送的内容, 第二部分总是一个空白行。这表示头部的终止以及内容的开始。

```
Content-type: text/html
```

该头部被 Show the Header 部分中的脚本发送出去。发送的内容包括设置标题栏文本和显示 H1 标题的初始 HTML 代码。

2. GET

`index.html` 包含两个表单, 如 16.2.6 节中的图 16-1 所示。第一个是 GET 表单, 它发送了两个名为 `one` 与 `two` 的文本输入, 还有 3 个名为 `check1`、`check2` 与 `check3` 的复选框。这些名称对网页源代码与 GET 请求的 URL 之外的用户而言是不可见的。表格将这些字段标记为 First Text、Second Text 与 Check These。表单被定义为 `action="/cgi-bin/hello.cgi" method="get"`。

`hello.cgi` 对 `QUERY_STRING` 变量进行分析, 并首先将所有的 `&` 符号转换成换行符。这样就会将 `one=hello+world&two=this+is+my+message&check1=on&check3=on` 转换为 4 个独立的行。它们由 `eval` 进行求值。这 4 行如下所示:

- `one=hello+world`
- `two=this+is+my+message`

- `check1=on`
- `check3=on`

注意，没有发送 `check2`，因为没有对它进行赋值。这对于复选框而言比较独特。如果第一个文本输入为空，则会以 `one=` 的形式发送，而且 `QUERY_STRING` 将为 `one=&two=this+is+my+message`。脚本随后将加号转换成空格(URL 不能包含空格)来显示消息。在下面的 16.2.6 节，图 16-2 展示了 `hello.cgi` 对所传递的文本与复选框的解析与显示。

这一技术本身具有固有的不安全性。向 `eval` 传递非受信用户的输入在任何情况下都非常危险。在下面的 16.2.6 一节，图 16-3 显示了恶意用户将变量名 `$DOCUMENT_ROOT` 输入到表单中。图 16-4 显示脚本看起来非常安全；美元符号已经被修改为没有危险的文本 `%24($` 的 ASCII 码为 `0x24`)。然而，这样的转换是由 Web 浏览器完成的，而浏览器是在用户的控制之下。图 16-5 显示用户可以修改地址栏中的 URL，且 Web 浏览器的文档根作为网页(在这里是 `/home/steve/book/web`)的一部分显示给恶意用户。该信息可能用来进行深入攻击。用户账户 `steve` 很可能存在于服务器中，并且可能开放了 `ssh`，或者在 Web 服务器上，甚至开放了 `ftp`。一旦我们可以让远程服务器为自己解释代码，我们便可以很容易地控制服务器上的一切。这样的信息并不是秘密，但我们不准再进行深入的介绍——本书并不是系统安全方面的书籍。

`hello.cgi` 的第二部分将美元符号转换为没有危险的 `$`。这意味着，图 16-5 的下半部分显示的是文本 `$DOCUMENT_ROOT`。它没有被脚本解释。对于任何可能的攻击而言，这依然不能保证绝对的安全，但它指出了一些需要防范的问题。

3. POST

在后面的 16.2.6 节中，图 16-6 再次显示了 `index.html` 页面，其中的第二个表单中包含了 `/etc/hosts` 与 `/etc/resolv.conf` 这两个本地文件名。POST 数据被从浏览器发送到脚本的标准输入。尽管可以使用 `method="POST"` 与 `action="upload.cgi?foo=bar"` 来向 CGI 脚本传递 GET 请求，但 POST 与 GET 这两个协议不能真正组合在一起。图 16-7 显示了两者的组合，不过这种组合不是很常用。POST 表单一般只使用 POST 元素，并忽略 `QUERY_STRING`。编码类型必须设置为 `multipart/form-data`，用来告诉浏览器文件需要为通过 HTTP 协议的发送而进行恰当编码。



通过将 `readfiles` 调用注释掉，以及对脚本末尾的 `<pre>` 标签与 `cat -` 命令的解注释，我们可以对该脚本进行测试。这样会显示出脚本接收到的原始数据。

发送的原始数据包括随机生成的边界字符串与每个文件之前包含文件名与变量名的头部。下面给出了一个输出示例。了解这个结构有助于对脚本解释进行理解。

```
-----92544452916948079257411075
Content-Disposition: form-data; name="fileone"; filename="hosts"
Content-Type: application/octet-stream
```

```

127.0.0.1      localhost www.example.com
192.168.1.3    router
192.168.1.5    plug
192.168.1.10   declan
192.168.1.11   atomic
192.168.1.13   goldie
192.168.1.227  elvis

-----92544452916948079257411075
Content-Disposition: form-data; name="filetwo"; filename="resolv.conf"
Content-Type: application/octet-stream

nameserver 192.168.1.3

-----92544452916948079257411075
Content-Disposition: form-data; name="filethree"; filename=""
Content-Type: application/octet-stream

-----92544452916948079257411075--

```

`upload.cgi` 脚本从读取第一行开始。第一行定义了边界字符串。边界字符串在文件之间起标记作用。它也被用在输出的末尾。末尾的`--`用来表示输出的结束。`boundary=${BOUNDARY%^\M}`一行将边界上结尾的 `Control-M` 去掉。这是有必要的，因为 Web 浏览器以 DOS 格式发送文本。DOS 格式在每行末尾有一个额外的`^\M`。这是 DOS 与 Unix 文本文件格式的区别。

`readfiles` 函数读取第一个文件的头部，然后在 `uploads` 子目录中创建一个空文件。子目录相对于文件系统中 `upload.cgi` 脚本的位置。随后 `while` 循环每次读取一行，将读取的上一行保存在 `$previous_line` 变量中。每行读取时都将`^\M` 去掉。`readfiles` 每次读取一行输入，并使用 `case` 语句判断下面的 3 种情况：

- 如果找到边界，则计算整个文件的 MD5 校验和，然后读取下一组头部。如果由于某些原因导致失败，则脚本退出。
- 如果发现边界之后有`--`，则计算上一次读取的文件的 MD5 校验和，然后函数返回。
- 对于所有其他情况，读取的上一行被追加到当前文件，然后将 `$previous_line` 变量赋值为最新读取的行。

使用 `$previous_line` 的原因在于，如果不使用的话，则对每组文件名头部的处理都会变得更加复杂。每个文本行的发送都伴随有一个空白行，然后在空白行之后发送边界字符串，所以如果 `case` 语句在接收到输入之后再执行一步则会显得更加明了。



在 `case` 语句的默认子句中有一个被注释掉的 `tee`。该项是可选的，用来在 Web 浏览器中显示上传文件的内容。显示文件内容不是该脚本的目的，但它可以用来说明只要对脚本进行小小的修改就能完成显示任务。

16.2.5 脚本代码

[illegible]

index.html

```
#!/bin/bash
echo "Content-type: text/html"
echo
```

```

cat - << EndOfHeaders
<html>
<head><title>Hello There!</title></head>
<body>
<h1>Hello There!</h1>
EndOfHeaders

echo "You said: $QUERY_STRING"

echo "<hr/>"
eval `echo ${QUERY_STRING} | tr '&' '\n'`
echo "one is ${one}" | tr '+' ' '
echo "<br/>"
echo "two is ${two}" | tr '+' ' '
echo "<br/>"
for check in check1 check2 check3
do
    if [ -z "${!check}" ]; then
        echo "${check} is not set<br/>"
    else
        echo "${check} is set<br/>"
    fi
done
echo "<hr/>"
eval `echo ${QUERY_STRING/'$'/'\$'} | tr '&' '\n'`
echo "one is ${one}" | tr '+' ' '
echo "<br/>"
echo "two is ${two}" | tr '+' ' '
echo "<hr/>"

cat - << EOF
</body>
</html>
EOF

```

hello.cgi

```
#!/bin/bash
```

```

function readfiles
{
    read disposition data name filename
    read ct contenttype
    read blankline
    read previous_line
    eval `echo $filename | tr -d '^M'`
    echo "<hr/>"
    echo "Processing file \"$filename\" ($contenttype)<br/>"
    > uploads/$filename
}

```

```

while read content
do
    contentvalue=${content%^M}
    case $contentvalue in
        $boundary)
            # end of file.
            # First, show the summary of the previous file
            cd uploads
            md5sum $filename
            cd - > /dev/null
            echo "<br/>"
            # Now read in the headers of the next file
            read disposition data name filename
            read ct contenttype
            read blankline
            read previous_line
            eval `echo $filename | tr -d '^M'`
            if [ ! -z "$filename" ]; then
                echo "<hr/>"
                echo "Processing file \"$filename\" ($contenttype)<br/>"
                > uploads/$filename
            else
                # That was the end of the input. No proper notification
                # received (boundary--) but handle it gracefully.
                echo "<hr/>"
                return
            fi
            ;;

        ${boundary}--)
            # end of all input
            cd uploads
            md5sum $filename
            cd - > /dev/null
            echo "<hr/>"
            return
            ;;
    *)
        echo "$previous_line" >> uploads/$filename # | tee -a uploads/$filename
        previous_line=$content
        ;;
    esac
done
}

# Show the Header
cat - << EndOfHeaders
Content-type: text/html

<html>

```

```

<head><title>Uploader</title></head>
<body>
<h1>File Uploads</h1>
EndOfHeaders

echo "Query String is $QUERY_STRING"

# Read the first line of input. This tells you the boundary
read BOUNDARY
boundary=${BOUNDARY%*M}

# Read and process the input
readfiles

# Use this instead for debugging and testing
# echo "<pre>"
# cat -
# echo "</pre>"

# Write the HTML footer
cat - << EOF
</body>
</html>
EOF

```

upload.cgi

16.2.6 调用结果

初始 Web 页面如图 16-1 所示。它显示了一个 GET 表单与一个 POST 表单。

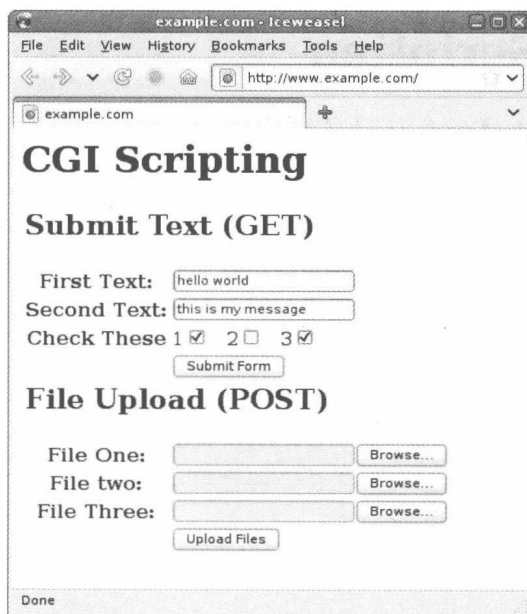


图 16-1

hello.cgi 处理的 GET 表单的结果如图 16-2 所示。这个简单的测试并没有试图对 CGI 脚本进行任何形式的欺骗，只是证明 CGI 脚本正常工作。

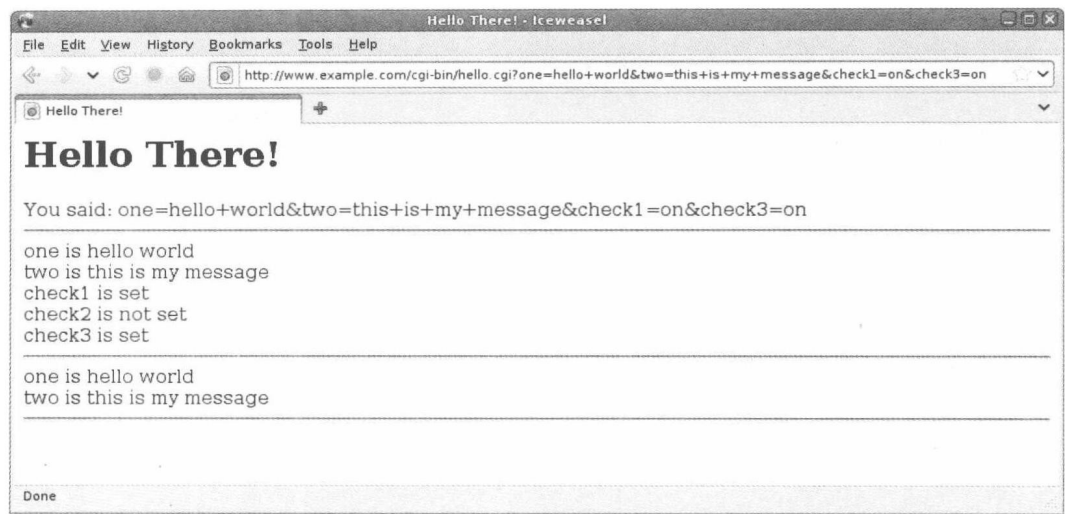


图 16-2

恶意用户向表单输入有害数据的情况如图 16-3 所示。这个简单的攻击本身不能起作用，因为 Web 浏览器在向 Web 服务器发送美元符号之前已对其进行了修改。

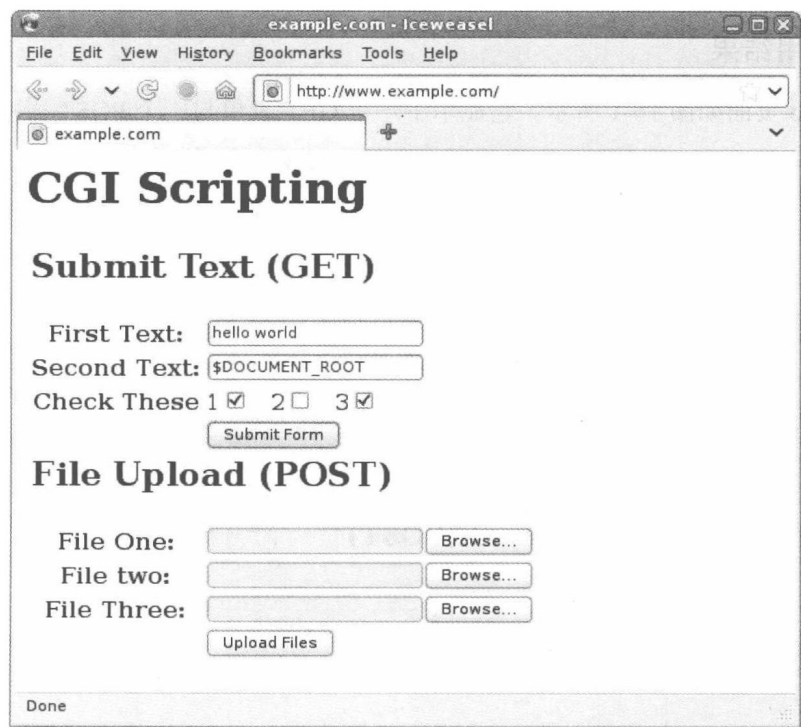


图 16-3

攻击结果如图 16-4 所示。美元符号被替换为无害的文本%24。



图 16-4

当用户通过直接修改地址栏中的 URL 对查询字符串进行编辑(将%24 改回美元符号)时所发生的情况如图 16-5 所示。这样一来，\$DOCUMENT_ROOT 的实际值被显示给攻击者。

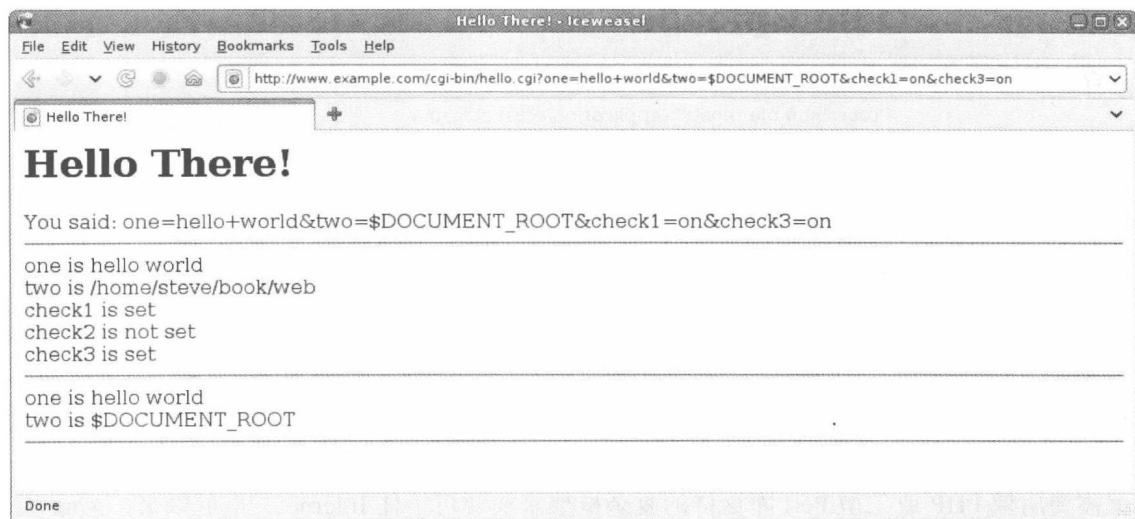


图 16-5

图 16-6 再次显示了初始 Web 页面。但页面经过了填充，准备通过 POST 表单向 Web 服务器发送两个文件。这些文件的名称与内容将作为 Web 请求的一部分发送出去。

在图 16-7 中，CGI 脚本显示了处理文件的更新过程，以及每个文件的 MD5 校验和。

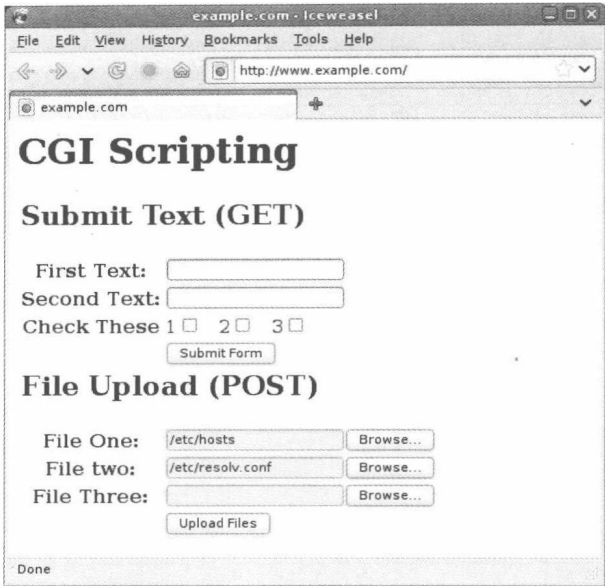


图 16-6

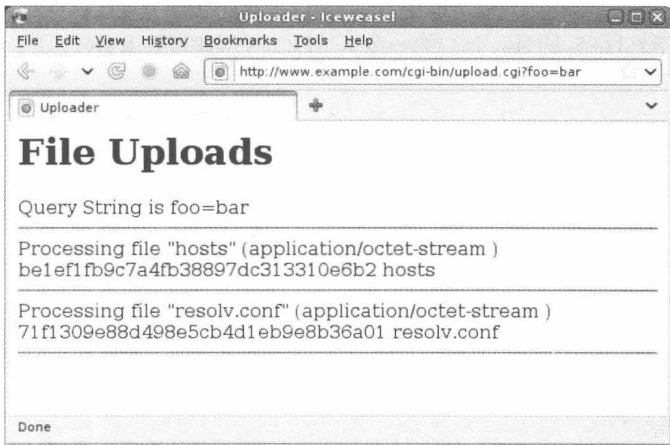


图 16-7

16.2.7 小结

CGI 是非常有用的工具，但它的设计所基于的假设是 Internet 是可信任的。这样一来就需要用像 PHP 或大型 Perl 库这样的复杂框架来实现可信任 Internet。最近以来，Ajax 大量取代 CGI 来以更加灵活与透明的方式提供服务器端的处理。shell 编写的 CGI 脚本仍然很有用，因为它们不需要 Web 服务器本身之外的任何第三方软件，并且可以非常快速与简单地组合到一起，还能提供非常重要的调试信息。

CGI 允许我们根据发送过来的输入对 Web 服务器的响应进行程序化的修改。有两种处理 CGI 脚本的方法：GET 更加透明，调试更简单；POST 允许向服务器发送文件。浏览器这一方面则将有 POST 表单转换为 GET 表单(与将 GET 表单转换为 POST 表单)的浏览器插

件。另外也可以使用 Wireshark 来跟踪 Web 流量以及调试 CGI 连接。

16.3 实用脚本 16-3: 配置文件

配置文件对于几乎所有的操作系统与应用程序都很常见。一些系统倾向于使用二进制文件，因为使用二进制文件可以更加容易地对应用程序的状态进行保存与读取。Unix 与 GNU/Linux 的传统是使用文本文件。这对于 shell 脚本的使用更加方便，同样意味着这些文件可以由人类来操作，手段可以是其他脚本与其他软件。也就是说，并不会因为数据或配置只能由软件的某一部分来读取，从而导致某个特定应用程序无法与其他程序之间进行交互。

16.3.1 用到的技术

- source(.)
- 变量赋值

16.3.2 概念

shell 脚本可以包含在单个文件中，其中包括所有的代码、数据与配置。然而，如果这些部分存储在单独的文件中，则脚本可以更加灵活。用户可以像简单的文本文件那样对配置进行编辑。这样一来，用户不必关心读取与解析这些配置的脚本的细节。这样的处理在所有的层次中都会有，从高层的应用程序用户到如编辑/etc/hosts 或/etc/sysconfig/network-scripts/ifcfg-eth0 这样的系统管理任务。这些文件本质上也是配置文件。

配置文件最简单的格式是使用与 shell 相同的语法。这样一来，我们可以使用 source(.) 命令直接将配置读取到环境中。这不是巧合。这是 shell 本身读取 ~/.profile、~/.bashrc 与其他配置文件的方式。

16.3.3 潜在的陷阱

这一技术的主要陷阱在于转义与引号字符可能存在问题。例如，如果变量被赋值为 a'b，则 shell 会一直读取到换行符之后，直到遇到一个对应的单引号为止。

```
$ cat eg.cfg
x=a'b
$ . eg.cfg
-bash: eg.cfg: line 1: unexpected EOF while looking for matching `''
-bash: eg.cfg: line 2: syntax error: unexpected end of file
$
```

16.3.4 脚本结构

该脚本显示了各种变量赋值的方法。在脚本的开头，它将变量置空。随后，如果配置文件存在且可读，则脚本使用 [-r \$CFG] 读取配置文件。read 中的 -p 开关包含了 read 命令

的提示符，所以 `read -p "Name: " name` 使用 “Name: ” 作为提示符读取 `name` 变量。

将(`$NAME`)添加到提示符文本中意味着，如果用户不输入值而按下回车键，则使用一个默认值。这是通过 `-z` 测试来实现的。它会测试变量是否为空，如果是，则将变量赋值为读取到的默认值。这样的映射遵循了大写与小写变量名的命名约定，所以 `$name` 是用户输入的名称，而 `$NAME` 是从配置文件读取的值。这并不是 `shell` 规定的标准，但这一命名约定很有用，因为可以很方便地拥有两个具有相同名称的不同变量。在后面的 16.3.6 节，`Bethany` 没有提供值，所以使用默认的 `Manchester`。

最后，脚本将当前值写入配置文件中。因为文件会被 `shell` 读取，所以标准的 `shell` 语法意味着以井号开头的注释会被忽略。

16.3.5 脚本代码



可从
wrox.com
下载源代码

```
DEBUG=0
NAME=Steve
LOCATION=Manchester

#!/bin/bash

DEBUG=3
NAME=
LOCATION=
COLOR=
CFG=`dirname $0`/name.cfg
[ -r $CFG ] && . $CFG

read -p "What is your name? ($NAME): " name
[ -z "$name" ] && name=$NAME
read -p "Where are you? ($LOCATION): " location
[ -z "$location" ] && location=$LOCATION
read -p "What is your favorite color? ($COLOR): " color
[ -z "$color" ] && color=$COLOR

echo "Hello ${name}, how is the weather in ${location}?"
echo "Can you see anything ${color}?"

echo "# Config file autogenerated by `id -nu` on `date`" > $CFG
echo "# Do not edit by hand, this file will be rewritten" >> $CFG
echo >> $CFG
echo DEBUG=$DEBUG >> $CFG
echo NAME=$name >> $CFG
echo LOCATION=$location >> $CFG
echo COLOR=$color >> $CFG
```

name.sh

name.cfg

16.3.6 调用结果

```
$ cat name.cfg
DEBUG=0
NAME=Steve
LOCATION=Manchester
$ ./name.sh
What is your name? (Steve): Bethany
Where are you? (Manchester):
What is your favorite color? (): Blue
Hello Bethany, how is the weather in Manchester?
Can you see anything Blue?
$ cat name.cfg
# Config file autogenerated by steve on Sun Apr 24 15:55:44 BST 2011
# Do not edit by hand, this file will be rewritten

DEBUG=0
NAME=Bethany
LOCATION=Manchester
COLOR=Blue
$ ./name.sh
What is your name? (Bethany): Emily
Where are you? (Manchester): the garden
What is your favorite color? (Blue): Pink
Hello Emily, how is the weather in the garden?
Can you see anything Pink?
$ cat name.cfg
# Config file autogenerated by steve on Sun Apr 24 15:56:07 BST 2011
# Do not edit by hand, this file will be rewritten

DEBUG=0
NAME=Emily
LOCATION=the garden
COLOR=Pink
$
```

16.3.7 小结

配置文件是存储默认值的很有用的方法。它们很容易读取，因为 `shell` 已经包含了对它们进行分析的代码。不需要额外的代码，就如同 Windows 中的 `.ini` 文件或其他一些预定的文件格式一样。这并不是组织配置文件的唯一方式，但却是最常用的，原因就是它很简单。

16.4 实用脚本 16-4：锁

让进程对某个资源具有独占式的访问权是非常有用的。例如，如果一个脚本需要花几分钟来处理与更新一个临界文件，处理时将其保持在一个非恒定的状态直到处理完成，这对于其他试图访问该文件的脚本而言是非常不好的，因为文件暂时处于一个已知的“坏”

状态。

锁系统能够提供这样一种机制，即在任何时刻只允许一个进程访问某个特定资源。资源本身可以是任何东西。锁只是防止多个进程同时做一件事情。这可以用来确保初始化脚本不会启动进程的多个副本(这种情况下经常使用/var/run/app-name.pid 作为锁文件)，或者其他任何只允许单独访问的任务。

在 shell 中实现锁定的一般方式是将运行进程的 PID 放到一个常规文件中。脚本的任何其他运行实例在进入临界代码部分之前对该文件进行检测，而且只有在没有其他运行实例声明锁的情况下继续运行。实际情况不是那么简单。进程必须考虑一种可能性的存在，那就是进程在检测了文件状态之后，锁文件可能已经被修改了。这一问题不能简单地通过状态的再次检测来修正，因为修改之后的状态也是未知的。



尽管该脚本在两个进程之间管理资源还算够用，但当有不止两个竞争进程存在时，具有潜在性的重叠的 sed 中会有 9 个系统调用。该脚本比在进程级别上的另外一种实现方式要安全很多倍，但如果没有专有的硬件支持，就不可能实现完全健壮的锁功能。

16.4.1 用到的技术

- 表示文件的原子修改的 sed -i
- 表示文件写入与追加的>与>>
- 文件系统一致性
- 循环

16.4.2 概念

锁定的概念相当简单。其实现稍微复杂一些，但也不是非常复杂。锁文件控制某些临界资源的访问权。在锁文件与资源本身之间并没有实际的连接。锁文件只是一种脚本用来确保其具有使用临界资源的时间间隙的自发性质的机制。

进程获取到锁，然后执行一些临界任务，再释放锁。当一个进程获得了锁，其他试图获取锁的进程会被限制在一个循环中，直到锁被释放。当原始进程释放锁后，试图获取锁的多个进程之间可能存在竞争。这一问题的解决方案是原子操作。尽管原子操作通常只被用于底层组件，如 CPU 微代码中实现的硬件测试与设置调用，但它也可以通过文件系统内部的一致性来实现。两个将各自的 PID(当然要小于文件系统的最小块长——一般为 8KB)写入相同文件中的不同进程不会相互影响。通过文件追加而不是覆盖，所有参与的进程都可以查看到锁的状态。锁是脚本实现实际锁机制方式的关键，在本章后面的 16.4.4 节会有解释。

16.4.3 潜在的陷阱

在 shell 中实现锁的问题在于获取锁的操作必须是所谓的原子过程，但 shell 中没有写

检查函数。如果脚本的两个运行实例发现锁处于可用状态，则两个实例都去要求获取锁，哪个会胜出？该脚本使用了一个 `while` 循环。该循环等待锁文件被删除，然后当锁可用时获取锁。



```
#!/bin/bash
LOCK=/tmp/myapp.lock

function get_lock
{
    MYPID=$1
    DELAY=2
    while [ -f "$LOCK" ]
    do
        sleep $DELAY
    done
    echo $MYPID > $LOCK
}

function release_lock
{
    rm -f $LOCK
}

echo "I am process $$"
get_lock $$
echo "$$: `date`" > /tmp/keyfile.txt
sleep 5
release_lock
cat /tmp/keyfile.txt
```

simplelock.sh

问题在于 `while` 循环的末尾。如果两个实例在同时运行，则调度器可能在任何时刻运行或暂停任何一个进程。这在大部分时间都没有任何问题，但有时会出现如表 16-2 所示的执行顺序。

表 16-2 非原子性锁

脚 本 1	脚 本 2	锁文件内容
	读取锁	空
读取锁		空
	写锁	PID #2
写锁		PID #1
写入临界文件		PID #1
	写入临界文件	PID #1

最后脚本的第二个实例会对临界文件进行写入操作，即使锁包含第一个实例的 PID。

这非常糟糕，而这正是锁系统能用来避免的问题。如果不能防止这种情况的发生，则整个锁机制也只是尽最大努力的解决方案，但不是完全健壮的。

缘于脚本的工作方式，在其中两个进程试图删除它们自己时，另一个进程可以从锁中将自己“删除”。如果第一个 `sed` 进程最后一个才完成(只有 9 个系统调用相互重叠)，则它可能将其他进程的 PID 写入锁中，这明显会使另外两个进程同时获取到锁。这两个进程可以继续执行，并且认为各自对锁都具有唯一的访问权。这在实际情况中很难发生，但对于没有写检查函数的硬件而言，这是不可避免的。

这里说明脚本的另一个不足：最后可能永远等待不会被释放的锁的访问权。这一问题可以使用 `timeout` 实用程序来回避，这样脚本至少还能退出并报告问题。然而，`timeout` 不适用于函数，而只能用于 `timeout`(本身作为外部程序)能执行的脚本与程序。这样做完全有可能，只是 `get_lock` 必须实现为一个独立的脚本，而不是函数。

16.4.4 脚本结构

这里包含了两个脚本：`domain-nolock.sh` 与 `domain.sh`。它们执行的任务相同。然而，`domain.sh` 使用锁来控制脚本的主体部分。这确保了最后输出的文件具有一致性。

`domain.sh` 显示了实际脚本完成的主要任务。这是一个非常简单的脚本，它对 Internet 域名进行 `whois` 查找，然后获取创建与过期日期，以及认证过的 DNS 服务器列表。它 3 次运行 `whois`，也就是 3 次独立的域名查询，这样做效率是较低的。这也可能导致 IP 地址暂时被封锁以防止进一步的域名查询。即使这里没有插入 `sleep` 语句用以保证明显的演示效果，这 3 个域名查询也要花掉大概 2 秒，所以最好是一次获取域名查询记录，然后对本地副本查询 3 次。

脚本使用 `tee -a` 向屏幕进行写操作，而且还会追加到日志文件。这会有锁系统导致的一些重要结果。



域名查询记录没有被很好地结构化，所以只对输出执行 `grep` 实际上不是获取信息的可靠方式。

`domain.sh` 脚本包含 3 个函数：`get_lock`、`release_lock` 与 `clean_up`。除了 `domain.sh` 首先调用 `get_lock`，然后在 `get_lock` 结束后调用 `release_lock` 以外，脚本的主体与 `domain-nolock.sh` 相同。

`get_lock` 是主要的函数。它等待锁被释放，然后将其 PID 添加到锁，再检查以确保是锁文件中唯一的 PID。锁的前一个拥有者可能没有清理锁文件就已经不存在了，所以其 PID 在锁文件中也是有可能的。如果代码到达这一阶段，则已经判断 PID 文件已被释放了。如果发现锁文件中有另一个 PID，则假设在竞争中失败。代码将 PID 从锁中删除、备份并再次尝试。脚本会在外层的 `while` 循环中执行，直到 `GOT_LOCK` 被设置为 1。

`get_lock` 的第一部分是循环 `while [-s "$LOCK"]`。该循环会等待直到锁文件为空或者不存在。如果满足该条件，则忽略整个 `while` 循环。如果发现锁文件且不为空，则从中读取 PID，并使用 `ps` 查找对应 PID 的进程名称。如果进程表中没有找到 PID 的进程名，则上

一个进程可能没有释放锁就已经结束了，所以循环用其他进程的 PID 调用 `release_lock` 从锁文件中删除 PID。释放 PID 而不是删除整个锁可以使得第三个进程能够注意到上一个 PID 已经结束，并能使这第三个进程获取到锁。当前实例没有进行任何假设，直到下次循环时发现文件为空时。如果第三个进程到这一阶段已经获取到锁，则该实例必须等待锁被释放。

如果锁文件中 PID 的进程依然在运行，则 `get_lock` 报告状态，并增加休眠的时延，然后再进入循环。根据应用程序的不同，适当的时延周期有所不同。定义一个上限可能比较有用。如果循环执行了 360 次，则时延将为 6 分钟，然后是 6 分 1 秒、6 分 2 秒等。

一旦处理完初始的 `while` 循环，锁文件中就不存在还在运行的进程了。脚本将自己的 PID 追加到锁文件中，然后检查锁文件中是否还包含其他内容。如果包含其他内容，则另外有一个进程在同一时刻做相同的操作。然而，因为 `grep` 对返回码赋值，所以 `grep` 与随后的 `if` 查询可以看成是原子性的。如果 `grep` 命令不能在文件找到另一个进程，则认为文件处于清理过的状态，且锁已经被获取。在操作系统级别上看，`grep` 不是一个原子命令。`grep` 进程本身的运行需要时间，而且在任何阶段都有可能被交换到交换分区。写后 `grep` 进程，加上文件系统的一致性可以确保其有效的原子性，因为无论如何都会有另一个运行实例让步。表 16-3 更详细地介绍了表 16-2 中的内容，以及在更坏情况下的反应。

表 16-3 原子性

脚 本 1	脚 本 2	锁文件内容
<code>echo \$\$ >> \$LOCK</code>		PID #1
<code>grep -vw \$MYPID \$LOCK(被中断)</code>		PID #1
	<code>echo \$\$ >> \$LOCK</code>	PID #1、PID #2
<code>grep(继续，返回失败；没有发现其他进程使用锁)</code>		PID #1、PID #2
	<code>grep -vw \$MYPID \$MYLOCK(成功；自上次 grep 开始，锁包含了两个 PID)</code>	PID #1、PID #2
<code>if</code> 调用检查 <code>grepd</code> 的返回码。如果失败，则获取到锁，即使其他脚本已经对锁文件进行了写操作		PID #1、PID #2
	调用 <code>if</code> 检查 <code>grep</code> 的返回码，然后删除自己的 PID	PID #1
将自己的 PID 写入锁文件。这并不是必要的		PID #1

如果 `grep` 命令成功，则意味着它找到了另一个试图获取锁的进程。`release_lock` 调用从锁文件中删除当前进程的 PID，然后进行一次随机长度的休眠(最长 5 秒)，尽量确保两个进程不会再次冲突，然后继续循环(因为 `GOT_LOCK` 还没有被赋值)。有可能两个运行实例都删除了锁中自己的 PID，但这不会导致任何问题。

`release_lock` 函数使用一个简单的 `sed` 命令从锁文件中删除一行。`$MYPID` 旁边的 `^` 与 `$`

分别表示行的开头与结尾，所以整行都必须匹配 PID。否则，从文件中删除 PID 123 也会删除其他无关的 PID 项，如 PID 1234。另一个从文件中删除一行的常见技术是使用 `grep -v $PID $LOCK > /tmp/tempfile.$$`，然后使用 `mv /tmp/tempfile.$$ $LOCK` 将新文件移回到原始文件的顶端。这完全不具有原子性，并且其他进程可能在 `grep` 与 `mv` 命令运行之间对文件进行了写操作。这样所做的变更就会丢失。虽然 `sed -i` 能有效地完成同样的任务，但其重叠部分处于系统调用级别，因此比这些进程要快几百倍。

如果脚本被中断，则调用 `cleanup` 函数。该调用通过 `trap` 将锁删除，条件是脚本在完成之前被终止。

16.4.5 脚本代码



可从
wrox.com
下载源代码

```
#!/bin/bash
KEYFILE=/tmp/domains.txt
MYDOMAIN=$1

echo "$MYDOMAIN Creation Date:" | tee -a $KEYFILE
sleep 2
whois $MYDOMAIN | grep -i created | cut -d":" -f2- | tee -a $KEYFILE
sleep 2
echo "$MYDOMAIN Expiration Date:" | tee -a $KEYFILE
sleep 2
whois $MYDOMAIN | grep "Expiration Date:" | cut -d":" -f2- | tee -a $KEYFILE
sleep 2
echo "$MYDOMAIN DNS Servers:" | tee -a $KEYFILE
sleep 2
whois $MYDOMAIN | grep "Name Server:" | cut -d":" -f2- | \
    grep -v "^$" | tee -a $KEYFILE
sleep 2
echo "... end of $MYDOMAIN information ..." | tee -a $KEYFILE
```

domain-nolock.sh

```
#!/bin/bash

# LOCK is a global variable. For this usage, lock.myapp.$$ is not suitable.
# /var/run is suitable for root-owned processes; others may use /tmp or /var/tmp
# or their home directory or application filesystem.
# LOCK=/var/run/lock.myapp
LOCK=/tmp/lock.myapp
KEYFILE=/tmp/domains.txt
MYDOMAIN=$1
mydom=/tmp/${MYDOMAIN}.$$

# See kill(1) for the different signals and what they are intended to do.
trap cleanup 1 2 3 6

function release_lock
{
```

```

MYPID=$1
echo "Releasing lock."
sed -i "/^${MYPID}$/d" $LOCK
}

function get_lock
{
    DELAY=2
    GOT_LOCK=0
    MYPID=$1

    while [ "$GOT_LOCK" -ne "1" ]
    do
        PID=
        while [ -s "$LOCK" ]
        do
            PID=`cat $LOCK 2>/dev/null`
            name=`ps -o comm= -p "$PID" 2>/dev/null`
            if [ -z "$name" ]; then
                echo "Process $PID has claimed the lock, but is not running."
                release_lock $PID
            else
                echo "Process $PID ($name) has already taken the lock:"
                ps -fp $PID | sed -e 1d
                date
                echo
                sleep $DELAY
                let DELAY="$DELAY + 1"
            fi
        done

        # Store our PID in the lock file
        echo $MYPID >> $LOCK

        # If another instance also wrote to the lock, it will contain
        # more than $$ and $PID
        # PID could be blank, so surround it with quotes.
        # Otherwise it is saying "-e $LOCK" and passing no filename,
        grep -vw $MYPID $LOCK > /dev/null 2>&1
        if [ "$?" -eq "0" ]; then
            # If $? is 0, then grep successfully found something else in the file.
            echo "An error occurred. Another process has taken the lock:"
            ps -fp `grep -vw -e $MYPID -e "$PID" $LOCK`
            # The other process can take care of itself.
            # Relinquish access to the lock
            # sed -i can do this atomically.
            # Back off by sleeping a random amount of time.
            sed -i "/^${MYPID}$/d" $LOCK
            sleep $((RANDOM % 5))
        else

```



```
GOT_LOCK=1
# Claim exclusive access to the lock
echo $MYPID > $LOCK
fi
done
}

function cleanup
{
    echo "$$: Caught signal: Exiting"
    release_lock
    exit 0
}

# Main Script goes here.
# You may want to do stuff without the lock here.

# Then get the lock for the exclusive work
get_lock $$

#####
# Do stuff #
#####
echo "$MYDOMAIN Creation Date:" | tee -a $KEYFILE
sleep 2
whois $MYDOMAIN | grep -i created | cut -d":" -f2- | tee -a $KEYFILE
sleep 2
echo "$MYDOMAIN Expiration Date:" | tee -a $KEYFILE
sleep 2
whois $MYDOMAIN | grep "Expiration Date:" | cut -d":" -f2- | tee -a $KEYFILE
sleep 2
echo "$MYDOMAIN DNS Servers:" | tee -a $KEYFILE
sleep 2
whois $MYDOMAIN | grep "Name Server:" | cut -d":" -f2- | \
    grep -v "^$" | tee -a $KEYFILE
sleep 2
echo "... end of $MYDOMAIN information ..." | tee -a $KEYFILE
echo >> $KEYFILE

# Then release the lock when you are done.
release_lock $$

# Again, there may be stuff that you will want to do after the lock is released
# Then cleanly exit.
exit 0
```

domain.sh

16.4.6 调用结果

下面的输出显示了两个不同的交互式 shell。第一个 shell 的提示符是 **Instance One**，且使用域名 **example.com** 调用 **domain-nolock.sh** 来进行查询。该 shell 运行完成，并且看上去一切正常。

```
Instance One$ ./domain-nolock.sh example.com
example.com Creation Date:
    1992-01-01
example.com Expiration Date:
    13-aug-2011
example.com DNS Servers:
    A.IANA-SERVERS.NET
    B.IANA-SERVERS.NET
... end of example.com information ...
Instance One$
```

第二个示例的提示符为 Instance Two，且使用域名 `steve-parker.org` 调用 `domain-nolock.sh` 来进行查询。看起来还是一切都很正常。

```
Instance Two$ ./domain-nolock.sh steve-parker.org
steve-parker.org Creation Date:
20-Jun-2000 13:48:46 UTC
steve-parker.org Expiration Date:
20-Jun-2011 13:48:46 UTC
steve-parker.org DNS Servers:
NS.123-REG.CO.UK
NS2.123-REG.CO.UK
... end of steve-parker.org information ...
Instance Two$
```

只有在输出文件被读回的时候问题才变得明显。两个脚本在同一时间向同一个文件进行写操作。现在文件内容一团糟。

```
Instance One$ cat /tmp/domains.txt
example.com Creation Date:
    1992-01-01
steve-parker.org Creation Date:
example.com Expiration Date:
20-Jun-2000 13:48:46 UTC
    13-aug-2011
steve-parker.org Expiration Date:
example.com DNS Servers:
20-Jun-2011 13:48:46 UTC
steve-parker.org DNS Servers:
    A.IANA-SERVERS.NET
    B.IANA-SERVERS.NET
NS.123-REG.CO.UK
NS2.123-REG.CO.UK
... end of example.com information ...
... end of steve-parker.org information ...

Instance One$
```

更好的解决方案是使用锁定。这一次，`domain.sh` 的调用使用了域名 `example.com`。还是运行完毕，且都很正常，除了最后的 `Releasing lock` 信息。

```
Instance One$ ./domain.sh example.com
example.com Creation Date:
    1992-01-01
example.com Expiration Date:
    13-aug-2011
example.com DNS Servers:
    A.IANA-SERVERS.NET
    B.IANA-SERVERS.NET
... end of example.com information ...
Releasing lock.
Instance One$
```

第二个示例也调用 `domain.sh`，并且使用域名 `steve-parker.org` 进行查询。该示例继续读取锁文件，然后休眠 2 秒、3 秒、4 秒、5 秒，直到锁被释放。第二个示例运行到临界代码中，向屏幕与输出文件进行写操作。输出文件不能由两个并发进程进行写入。

```
Instance Two$ ./domain.sh steve-parker.org
Process 14228 (domain.sh) has already taken the lock:
Steve 14228 12786 0 12:47 pts/7 00:00:00 /bin/bash ./domain.sh example.com
Fri Apr 22 12:47:11 BST 2011

Process 14228 (domain.sh) has already taken the lock:
Steve 14228 12786 0 12:47 pts/7 00:00:00 /bin/bash ./domain.sh example.com
Fri Apr 22 12:47:14 BST 2011

Process 14228 (domain.sh) has already taken the lock:
Steve 14228 12786 0 12:47 pts/7 00:00:00 /bin/bash ./domain.sh example.com
Fri Apr 22 12:47:17 BST 2011

Process 14228 (domain.sh) has already taken the lock:
Steve 14228 12786 0 12:47 pts/7 00:00:00 /bin/bash ./domain.sh example.com
Fri Apr 22 12:47:21 BST 2011

steve-parker.org Creation Date:
20-Jun-2000 13:48:46 UTC
steve-parker.org Expiration Date:
20-Jun-2011 13:48:46 UTC
steve-parker.org DNS Servers:
NS.123-REG.CO.UK
NS2.123-REG.CO.UK
... end of example.com information ...
Releasing lock.
Instance Two$
```

现在，输出文件被清晰地分为两部分：第一部分是 `example.com` 的细节，而第二部分

是 `steve-parker.org` 的细节。

```
Instance One$ cat /tmp/domains.txt
example.com Creation Date:
    1992-01-01
example.com Expiration Date:
    13-aug-2011
example.com DNS Servers:
    A.IANA-SERVERS.NET
    B.IANA-SERVERS.NET
... end of example.com information ...

steve-parker.org Creation Date:
20-Jun-2000 13:48:46 UTC
steve-parker.org Expiration Date:
20-Jun-2011 13:48:46 UTC
steve-parker.org DNS Servers:
NS.123-REG.CO.UK
NS2.123-REG.CO.UK
... end of steve-parker.org information ...

Instance One$
```

16.4.7 小结

锁是确保代码的运行实例与其他运行实例被区别对待的有用方法，因为只有一个运行实例持有锁。一旦持有了锁，则独占权限不再有任何限制。可以写文件(如脚本所示)或者用来确保只有进程的一个实例运行。只要希望确保正在进行某个特定操作的进程在运行时不会受到其他进程的影响，我们就可以使用锁。

如调用示例所示，无论使用或者不使用锁，第一个进程都会继续运行，而不知道是否有另一个进程在运行，更不用说另一个进程希望访问第一个进程正在使用的同一个资源。两者的区别是，使用锁的版本保证了资源的独占性访问。

第 17 章

演 示

演示可以让用户对 shell 脚本留下深刻的印象，并且还能使其易于使用。不是所有的 shell 脚本都只是用来完成一次性的任务。即使经年累月，一些脚本还是会被用到，而且用户群还非常大。脚本不仅仅局限于将白色的文本向黑色的背景上一行行地输出。本章展示了 shell 脚本的一些功能，并且没有使用花哨的技巧，也不依赖于其他子系统。

17.1 实用脚本 17-1：太空游戏

该脚本的灵感来自 20 世纪 70 年代的经典街机游戏《太空入侵者》。游戏的目标是在外星人到达地球(用屏幕的底部表示)之前将其歼灭。我们可以使用 a 与 l 键控制飞船的左右移动，使用 f 键来开火。每次只发一枚炮弹。

17.1.1 用到的技术

- kill、trap 以及用于定时的 SIGALRM
- 及时响应键盘输入的高级 read 用法
- 控制终端的 tput
- 用于显示的 ANSI 颜色
- 数组，尤其是向函数传递数组
- 计算位置与进行碰撞检测的一些基本数学方法

17.1.2 概念

游戏本身所包含的概念非常简单。外星人从左向右移动，然后再反向，并且向屏幕底端下降，所以每当它们到达屏幕的右端就向下移动一个文本行。这是通过每次增加 ceiling 变量的值来实现的。外星人显示的位置是(row*2)+ceiling，乘以 2 表示每一波外星人之间有一个空白行，将该值加到一直在增大的 ceiling 表示整个外星人军队随时间慢慢靠近地面。

我们有一挺激光炮，它用一对变量(cannonX 与 cannonY)表示。这两个变量用来跟踪激

光炮的位置。`cannonX` 是必需的，因为激光即使在我们移动了位置之后仍保持垂直方向的移动，所以激光发射的位置与飞船的位置相同，但发出之后与飞船不再相关。这样的实现意味着直到上一枚炮弹击中敌人或到达屏幕的顶端才能进行第二次发射。通过实现 `(cannonX,cannonY)` 变量对的数组，我们可以改进这一不足，但这样不忠实于原游戏，并且使得用激光消灭敌人变得很容易。

数据结构非常简单。飞船与激光炮使用简单的整数变量来关联它们的位置。每行外星人在内部使用数组表示。数组在外星人被击中后保存得分。该数组还用来跟踪残留下来的外星人——当外星人被消灭时，其数组中的值加入到得分中，然后数组中的值被降到 0。`drawrow` 函数将 0 理解为该位置没有外星人，所以不用显示，且在碰撞检测时也不用考虑。

`aliens1` 与 `aliens2` 数组用来存储每行外星人外形的编码。每行都有各自的颜色与形状，用 ASCII 字符画表示。通过使用这两个不同的数组与模 2 操作 `if (($offset % 2 == 0))`，外星人在屏幕中的移动会比简单地从一个位置移动到另一个位置更具动画效果。

`shell` 不适合写这样的游戏的两个因素是，在不需要用户按键之后再按回车键的情况下对按键的交互式实时读取，以及屏幕的频繁刷新。早期的 Unix 系统中没有 `read -n` 语法，但 GNU 系统与 Solaris 10 提供这种语法。对外星人的实时更新是通过 `SIGALRM` 信号实现的。该信号每隔 `$DELAY` 秒就唤醒脚本对显示进行更新。每次脚本被唤醒，`move` 函数都会设置 `$DELAY` 秒之后的下一次唤醒，与按下闹钟上的“打盹”按钮有几分相似。

脚本中的 `sleep $DELAY` 假设 `sleep` 可以接受非整数数值。对 GNU 的 `sleep` 而言可以，而传统 Unix 的 `sleep` 则不行，后者不能比休眠 1 秒的 `sleep 1` 更短。不过，这使得游戏在 Unix 中运行非常缓慢。可以避免这一不足，那就是每 `X` 次迭代进行一次休眠，就好像 `$DELAY` 逐渐减小，`X` 逐渐增大。这留给读者作为练习。

因为读取键盘输入的循环独立于 `move` 函数的定时调用，所以我们可以得到关于飞船移动的反馈，并且独立于外星人的位置更新。这种独立性对于游戏的交互感非常重要，使游戏不像是基于回合制。

关于 `bash` 中的数组比较让人沮丧的是它无法作为参数传递给函数，也无法作为返回值返回。有一种解除这一限制的方法。使用 `"${a[@]}"` 调用函数，然后处理 `"$@"` (引号在前后都是必需的) 就可以得到数组元素，并且保留了数组元素中的空格。



可从
wrox.com
下载源代码

```
$ cat func-array.sh
#!/bin/bash

a=( one "two three" four five )

function myfunc
{
    for value in "$@"
    do
        echo I was passed: $value
    done
}
```

```
myfunc "${a[@]}"
```

```
$ ./func-array.sh
I was passed: one
I was passed: two three
I was passed: four
I was passed: five
$
```

[func-array.sh](#)

函数无法将数组作为数值返回。尽管可以返回基本数值，但这种方法不具有健壮性，并且对空白字符不予处理。下面的代码作为参考，**bash** 也可以对其进行管理，但是作用有限。



可从
wrox.com
下载源代码

```
$ cat array-func.sh
#!/bin/bash

a=( one two three four five )

function myfunc
{
    declare -a b
    i=0
    for value in "$@"
    do
        b[i]="abc.${value}.def"
        ((i++))
        shift
    done
    echo "${b[@]}"
}

for value in "${a[@]}"
do
    echo "Item is $value"
done
declare -a c
c=(`myfunc "${a[@]}"`)
for value in "${c[@]}"
do
    echo "Converted Item is $value"
done

$ ./array-func.sh
Item is one
Item is two
Item is three
Item is four
Item is five
```



```

Converted Item is abc.one.def
Converted Item is abc.two.def
Converted Item is abc.three.def
Converted Item is abc.four.def
Converted Item is abc.five.def
$

```

array-func.sh

17.1.3 潜在的陷阱

正确地进行碰撞检测会有些困难，尤其是外星人宽度大于一个单元格的时候。让屏幕保持干净也需要谨慎进行。对屏幕进行太多刷新会毁掉游戏，因为会导致过度闪烁。相对而言，调用 `clear` 命令会花较长时间，并且使屏幕非常闪烁。

编写代码时，该脚本的最显著的改变在于 `for` 循环中模功能从函数调用转移为外部 `expr` 命令。这意味着对每个外星人(无论死活)调用 `expr`，然后替换为内嵌的 `((... %2))` 结构。从 `expr` 到内嵌方法的修改意味着可以改变外星人的形状。不修改的话，执行会非常缓慢。将模值从 `for` 循环中取出也会效率更高一些，但不会特别明显。

17.1.4 脚本结构

脚本的开头与结尾是 `tput` 命令。它可以使光标在游戏开始前消失(`tput cinvis`)，然后在退出后又出现(`tput cvvis`)。另一个较小的修改是在游戏结束时取消 `SIGALRM` 上的 `trap`，这样 `move` 函数在脚本结束后就不会发送 `SIGALRM` 信号。这些只是小细节，但可以对效果有相当大的改观。否则，如果没有它们，效果会差很多。

该脚本由 4 个关键函数与一个主循环构成。从脚本的底部往上，主循环只从键盘读取一个字符(`read -n 1`)。如果是“左”或“右”指令(分别为 `a` 和 `l`)，则循环更新飞船的位置。飞船将立刻被重新绘制。如果按下的是开火按钮(`f`)，且激光炮不处于使用状态(`cannonY -eq 0`)，则 `cannonX` 变量相对于飞船当前的 X 轴坐标进行赋值，`cannonY` 被赋值为飞船的 Y 轴坐标(固定在屏幕的底部)。

只要按键移动了飞船，主循环就调用 `drawship`。该函数用 `printf` 语句清空屏幕的整个底行，并将飞船中的激光炮用彩色显示，用以表示是否装配了火炮。这与 `move` 函数是独立的，并且能实时地更新飞船的移动，且与单调缓慢的外星飞船的更新不同。`drawship` 也在 `move` 函数中调用，这样即使在飞船没有移动的情况下，火炮的更新也能正常反映出来。

主循环的开头是 `move` 函数。`move` 函数使用 `SIGALRM` 在 `$DELAY` 秒之后调用自身。`DELAY` 会随着时间推移越来越小，所以外星人下降的速度越来越快。每次调用 `move` 时，外星人在它们的前进方向上移动一格。当它们到达屏幕的边界时，就翻转 `direction` 变量使它们向相反的方向运动。外星人在每次抵达屏幕的右边时都会向下移动一行(通过增加 `ceiling` 变量)。

`move` 随后对每行外星人调用一次 `drawrow` 函数。因为 `bash` 没有多维数组，所以外星人的行数被硬编码到脚本中。使用循环来遍历所有行是一个不错的选择，就算有 6 行外星人，6 次调用 `drawrow` 函数也不会显得太笨拙。在显示过程中，`drawrow` 返回碰到炮弹的所有外星人的索引号，如果当前行中没有外星人被击中则返回 0。这样对数组的结构产生

了一些额外的效果。数组从 0 开始索引，但因为 0 在 `drawrow` 函数的返回码中具有特殊意义，所以保存外星人分值的数组 `row0` 到 `row5` 不使用 0 号索引。这或许有些拙劣，但可以使每一行在调用 `drawrow` 之后进行直接赋值 `rowX[$?]=0`。如果 `$?` 为 0，则更新未使用的 0 号索引，对其他数组元素没有影响。如果 `$?` 大于 0，则它会指向某个外星人，然后将存储其得分值的数组元素赋值为 0。这样一来，随后对 `drawrow` 的调用不会在原来位置显示外星人，并且碰撞检测会允许炮弹通过该位置而不停下来。更简洁的实现方法要求对 `drawrow` 的每次调用之后都有一些更加复杂的代码。这些代码会检查返回值，并且在外星人被击中的情况下更新数组。该脚本使用的方法使得游戏在最低限度上能快速运行，并且更重要的是易于阅读与管理。

接着对残留的外星人计数。如果扫除了所有的入侵者，则显示祝贺的消息并退出游戏。游戏很可能会继续进行，且下一波的入侵会更快。这很容易实现，但会增加一些复杂度，且代码会更长一些。

最后，`move` 调用 `drawcannon` 函数。可以将 `drawcannon` 函数的功能写在 `move` 函数中，但将它抽象到自己的函数中会更加清晰一些。`drawcannon` 只是在前一个位置的上方显示一个空白字符，然后计算新的位置(比之前高一格的位置)，最后在新位置重新显示炮弹。

`drawrow` 函数所做的工作最多。除了显示外星人，它还要进行碰撞检测来判断是否击中外星人。第一个参数告诉该函数要显示的外星人的种类，余下的参数则是 `rowX()` 数组中适当的数值。之前已经介绍过，向函数传递数组比较困难，且不是唯一可行的方法。但是这样一来代码会比较简单。`shift` 得到 `alientype` 参数：`$@` 的余下部分则是告诉 `drawrow` 哪些外星人未被击中，以及它们的得分值的数值序列。这样的实现方式同样为外星人能够承受多次打击的构思留下很大的余地。如果 `drawrow` 没有将数值减小到 0，则可以只从外星人的生命值中减掉一个固定值。余下的生命值可能大于 0，这样外星人在后续的运行中还是会出现，直到它被彻底击毁。

`drawrow` 函数首先查看本行是否有外星人。如果没有外星人，则该行没有外星人入侵，所以函数在执行进一步测试之前就退出了。如果有外星人，则计算这一行要显示的高度。如果高度与 `$bottom` 变量(定义我们的飞船固定所在的 Y 轴坐标)相等，则入侵成功，游戏失败，然后以一条适当的消息退出。否则脚本继续执行。

对于每个存在的外星人，它所占据的空间会与激光炮弹的位置进行比较。如果它们相等，则使用 3 个红色的星号代替外星人的显示图标(定义在 `$avatar` 变量中)，用来表示击中后产生的爆炸效果。我们使用存储在数组中的数值对玩家进行加分，且用 `$skilled` 变量保存数组中被击中外星人的索引号，以便在调用 `draw` 函数时将其清零。这就不需要 `drawrow` 知道要更新的数组名。假设 `bash` 支持多维数组，则没有必要这么做。但由于每一行都是独立的数组，我们很容易让 `drawrow` 向调用者返回索引号，并让调用者维护其状态，以及用被消灭的外星人的索引号对正确的数组进行更新。最后显示外星人(或者相应的空间)，然后循环接着显示行内的下一个外星人。

17.1.5 脚本代码

```
#!/bin/bash
stty -echo

# Make the cursor invisible (man terminfo)
```

```
tput civis
clear

cat - << EOF
```

```
SPACE
```

```
LEFT:      a
RIGHT:     l
FIRE:      f

QUIT:      q
```

```
Press any key.
```

```
EOF
```

```
read -s -n 1
```

```
row0=( 0 30 30 30 30 30 30 30 30 )
row1=( 0 20 20 20 20 20 20 20 20 )
row2=( 0 15 15 15 15 15 15 15 15 )
row3=( 0 10 10 10 10 10 10 10 10 )
row4=( 0 5 5 5 5 5 5 5 5 )
row5=( 0 1 1 1 1 1 1 1 1 )
```

```
aliens1=( '\033[1;32m|0|\033[0m' '\033[1;34m\~/\033[0m'
          '\033[1;35m:x:\033[0m' '\033[1;38m:#:\033[0m'
          '\033[1;33m|!!\033[0m' '\033[1;39m:-:\033[0m' ]
aliens2=( '\033[1;32m:0:\033[0m' '\033[1;34m/-\\033[0m'
          '\033[1;35m-x-\033[0m' '\033[1;38m-#-\033[0m'
          '\033[1;33m:|:\033[0m' '\033[1;39m-:-\033[0m' ]
```

```
score=0
```

```
# farthest right that the *leftmost* alien can go to
MAXRIGHT=46
# furthest right that the ship can go to
FARRIGHT=73
```

```
# Ship's current position (x-axis)
ship=30
# Cannon column; remains the same even if ship moves
cannonX=$ship
# Cannon height; 0 means it's ready to fire
cannonY=0
# Positive direction to right, Negative to left
direction=1
offset=20
bottom=20
```

```

ceiling=4
MAXCEILING=6
DELAY=0.4

function drawrow
{
    # draw a row of aliens; return the index of any alien killed
    # note that only one alien can be killed at any time.
    alientype=$1
    shift
    let row="$alientype * 2 + $ceiling"
    aliensonrow=`echo $@ | tr ' ' '+' | bc`
    if [ $aliensonrow -eq 0 ]; then
        # Nothing to do here. In particular, do not detect failure.
        # Just clear the previous line (it may contain the final explosion
        # on that row) and return.
        tput cup $row 0
        printf "%80s" " "
        return 0
    fi
    if [ $row -eq $bottom ]; then
        tput cup `expr $bottom - 4` 6
        trap exit ALRM
        echo "YOU LOSE"
        sleep $DELAY
        stty echo
        tput cvvis
        exit 1
    fi
    declare -a thisrow
    thisrow=( `echo $@` )

    tput cup 0 0
    printf "Score: %-80d" $score

    killed=0
    # Clear the previous line
    tput cup `expr $row - 1` 0
    printf "%80s" " "

    tput cup $row 0
    printf "%80s" " "
    tput cup $row 0
    printf "%-${offset}s"

    # Don't do this calculation in the for loop, it is slow even without expr
    if (( $offset % 2 == 0 )); then
        thisalien=${aliens1[$alientype]}
    else
        thisalien=${aliens2[$alientype]}
    fi
}

```

```
fi

# there are 8 aliens per row.
for i in `seq 1 8`
do
    value=${thisrow[$i]}
    avatar=$thisalien

    if [ $value -gt 0 ]; then
        # detect and mark a collision
        if [ $row -eq $cannonY ]; then
            let LEFT="$i * 4 + $offset - 4"
            let RIGHT="$i * 4 + $offset - 1"
            if [ $cannonX -ge $LEFT ] && [ $cannonX -le $RIGHT ]; then
                killed=$i
                avatar='\033[1;31m***\033[0m'
                ((score=$score + $value))
                cannonY=0
            fi
        fi
    fi

    if [ $value -eq 0 ]; then
        printf "  "
    else
        echo -en "${avatar} "
    fi
done
return $killed
}

function drawcannon
{
    # move the cannon up one
    if [ $cannonY -eq 0 ]; then
        # fell off the top of the screen
        return
    fi

    tput cup $cannonY $cannonX
    printf "  "
    ((cannonY=cannonY-1))
    tput cup $cannonY $cannonX
    echo -en "\033[1;31m*\033[0m"
}

function drawship
{
    tput cup $bottom 0
    printf "%80s" "  "
```

```

tput cup $bottom $ship
# Show cannon state by its color in the spaceship
if [ $cannonY -eq 0 ]; then
    col=31
else
    col=30
fi
echo -en "|--\033[1;${col}m*\033[0m--|"
}

function move
{
    # shift aliens left or right
    # move cannon, check for collision

    (sleep $DELAY && kill -ALRM $$) &

    # Change direction if hit the side of the screen
    if [ $offset -gt $MAXRIGHT ] && [ $direction -eq 1 ]; then
        # speed up if hit the right side of the screen
        DELAY=`echo $DELAY \* 0.90 | bc`
        direction=-1
        ((ceiling++))
    elif [ $offset -eq 0 ] && [ $direction -eq -1 ]; then
        direction=1
    fi

    ((offset=offset+direction))

    drawrow 0 ${row0[@]}
    row0[$?]=0
    drawrow 1 ${row1[@]}
    row1[$?]=0
    drawrow 2 ${row2[@]}
    row2[$?]=0
    drawrow 3 ${row3[@]}
    row3[$?]=0
    drawrow 4 ${row4[@]}
    row4[$?]=0
    drawrow 5 ${row5[@]}
    row5[$?]=0

    aliensleft=`echo ${row0[@]} ${row1[@]} ${row2[@]} ${row3[@]}\
        ${row4[@]} ${row5[@]} \
        | tr ' ' '+' | bc`
    if [ $aliensleft -eq 0 ]; then
        tput cup 5 5
        trap exit ALRM
        echo "YOU WIN"sleep $DELAY      tput echo
        tput cvvis
    fi
}

```

```
        echo; echo; echo
        exit 0
    fi

    drawcannon
    drawship
}

trap move ALRM

clear
drawship
# Start the aliens moving...
move
while :
do
    read -s -n 1 key
    case "$key" in
        a)
            [ $ship -gt 0 ] && ((ship=ship-1))
            drawship
            ;;
        l)
            [ $ship -lt $FARRIGHT ] && ((ship=ship+1))
            drawship
            ;;
        f)
            if [ $cannonY -eq 0 ]; then
                let cannonX="$ship + 3"
                cannonY=$bottom
            fi
            ;;
        q)
            echo "Goodbye!"
            tput cvvis
            stty echo
            trap exit ALRM
            sleep $DELAY
            exit 0
            ;;
    esac
done
```

17.1.6 调用结果

游戏用常规方式运行。如果使用彩色终端，则应当是全彩色的，且闪烁非常小。游戏画面如图 17-1 所示。

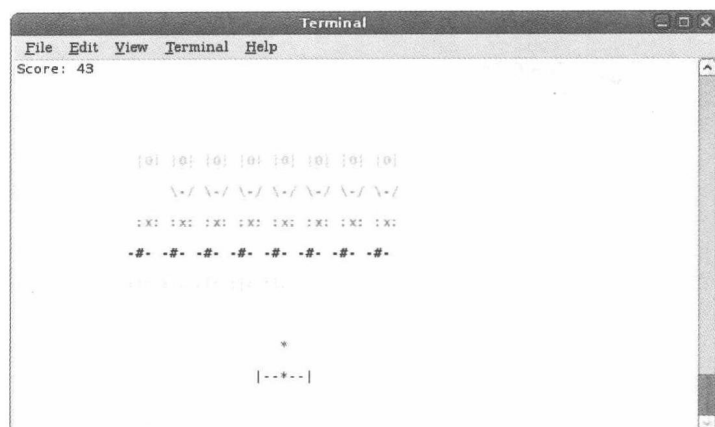


图 17-1

图 17-2 显示外星人飞船被击中后的爆炸效果。3 个红色星号表示爆炸。

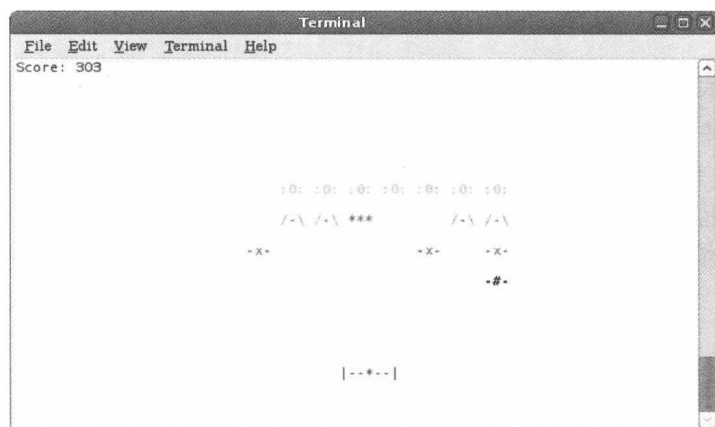


图 17-2

图 17-3 显示外星人飞船抵达地球后游戏结束。

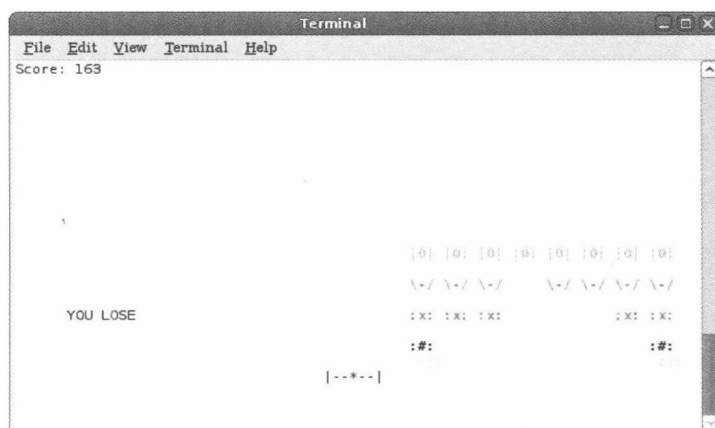


图 17-3

图 17-4 显示玩家歼灭所有来犯者后赢得游戏。

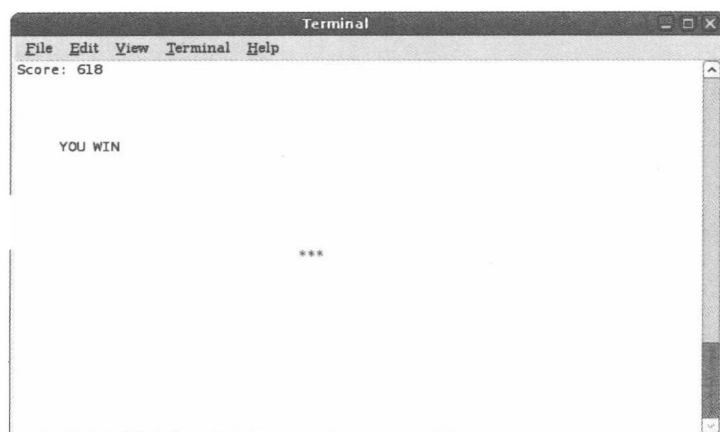


图 17-4

17.1.7 小结

尽管趣味性不大，但该脚本证明了可以用 `shell` 脚本完成非常具有交互性的任务，而不仅仅是普通的“按 1 继续，按 2 退出”式的菜单系统。即便如此，后者还是通常被认为是交互式的 `shell` 脚本。`shell` 可以完成一般功能之外更多的任务。尽管 `shell` 要成为游戏开发的理想语言还远远不够，但希望该脚本可以激发更多 `shell` 脚本的使用灵感，尤其是 `bash` (如数组) 与 GNU 环境 (如可以进行亚秒级的休眠，否则该游戏的趣味性将大打折扣) 中一些新的特性。

第 18 章

数据存储与检索

数据的检索、处理与存储就是计算的全部。本章介绍两个脚本来完成计算任务。第一个脚本处理 HTML 文档来识别与使用包含在文档中的所有链接。实现这一点其实不容易，所以该脚本介绍了一些必要的操作，它们用来确保没有漏掉链接，且普通文本不会被误认为是链接。第二个脚本从 Linux 内核的 `/proc` 虚拟文件系统中分析内核状态，然后将其转换为 CSV 格式。CSV 格式可以由电子表格软件分析并生成图表。

两个脚本旗鼓相当，因为第一个读取的数据将被像 Web 浏览器这样的图形化桌面软件读取。第二个创建的数据可以被桌面电子表格软件解析。尽管完全是基于文本的，但 shell 在一定程度上可以分析以及为图形化软件产生数据。

18.1 实用脚本 18-1：分析 HTML

HTML 是非常常见的标记语言。但有很多 HTML 在编写上非常糟糕，这使得对这样的文件进行分析相当困难。本脚本中的结构从 HTML 中截取标签(`<a>`、``等)。`downloader.sh` 脚本作用于 `<a>` 标签，即将链接的 URL 保存到以锚文本命名的文件中。像 `This is an example web site` 这样的输入会将 `www.example.com` 的索引页下载到名为 `This is an example web site` 的文件中。

18.1.1 用到的技术

- `tr`
- `((suffix++))`
- `wget`

18.1.2 概念

该脚本实际所做的工作并不是作用很大。`wget -Fi` 完成与该脚本非常类似的任务，但该脚本主要是介绍从 HTML 输入中截取标签。

一些 HTML 术语被用在该脚本中。在输入 `example pages` 中，`/eg.shtml` 是链接，`example pages` 是锚文本。在默认情况下，锚文本在浏览器中用蓝色下划线表示，链接是点击锚文本后所显示的页面的地址。

该脚本使用一个简单的状态机来跟踪截取操作所在的 HTML 输入的位置。如果没有状态机，我们就必须对输入文件的格式进行更多的假设。

18.1.3 潜在的陷阱

HTML 的处理存在很多陷阱。尽管现在的大多数 HTML 都是 HTML 4.01、XHTML 或是一堆未验证的标签(相当于 HTML 3，但不完全准确)，但 HTML 没有一个单一的定义。该脚本试着以一种合理的方式将输入组织到一组标签中，但还是要对组织结构进行很多隐性的假设。如果输入文件可以由 Web 浏览器呈现，则脚本也应当能够成功地将链接截取出来。

18.1.4 脚本结构

脚本开头删除一些关键变量，然后设置下载目录。如果下载目录不存在则进行创建。INFILE 变量被设置为 `${HOME}/.mozilla/firefox/*/bookmarks.html`，说明变量在 `${1:-default}` 语法中也可以使用。

`tr` 语句列表将输入组织成一个长行的 HTML 文件，然后将 `<` 与 `>` 符号转换成换行符。在换行与空格之后，其他种类的空白字符为制表符。所以它们被转换为空格，然后 `tr -s ' '` 将空格序列压缩成一个空格。这就是 HTML 期望的空白字符处理方式。这会有一个副作用，即包含多个空格的锚文本也会被压缩；Web 浏览器对这样的输入也是这样处理。所以 `example pages` 会变成 3 行输入。

- `a href="/eg.shtml"`
- `example pages`
- `/a`

`while` 语句可以用来找到被截断成多行的输入。`while` 语句的主体是单个 `if ... else` 语句。我们首先介绍其中的 `else` 部分。`else` 部分检测标签是否为 `a`(或 `A`)，并检测状态机的当前状态是否还处于锚文本中(因为如注释所描述的那样，`a href` 即使看起来不对，但也是合法的)。锚文本可以是 `a welcome page`，但仅仅因为第一个单词为 `a` 不能得出它就是链接的开头这样的结论。如果标签为 `a`，且正在处理的不是锚文本，则处理的是链接。

对链接本身的解析也可能比较棘手。这里以 Firefox 书签为例，它包含了像 `A HREF="http://fxfeeds.mozilla.com/en-US/firefox/livebookmarks/" FEEDURL="http://fxfeeds.mozilla.com/en-US/firefox/headlines.xml" ID="rdf:#$HvPhC3"` 这样的链接。为了从以上内容中截取正确的部分，脚本再次将输入分解成多行，然后选取包含 `href` 的那一行。随后删除前 7 个字符(删除 `a href=`)，再删除 URL 两旁应有(但也不总会有)的引号。最后，如果得到的链接包含任何文本，则将 `$state` 变量设置为 `anchor`。

`if ... else` 语句的另一部分处理另一种情况。`if` 语句测试 `$state` 是否为 `anchor` 且 `$tag` 不为空。如果标签为 `img`，则锚“文本”实际上是一个图像。将图像信息保存为文件名不太可能，所以脚本会保存为 `img.1`、`img.2` 等。脚本直接将行中的余下部分删除，所以 `img`

src="/example.png"变成.

接下来, 如果锚文本是多个单词, 则\$label 变量会包含第一个单词之后的余下单词, 所以有 filename="\$tag \$label"。不然锚文本就包含唯一一个单词, 所以有 filename=\$tag。脚本随后检查文件是否存在于下载目录中。如果存在, 则检查\${filename}.1、\${filename}.2 等。我们使用一个简单的 while 循环来追加后缀的数目以及增加后缀数值, 直到\$filename 变量包含的是一个不存在文件的名称。

最后, 脚本在读取到链接后将状态设置为 anchor, 并读取锚文本。脚本调用 wget 下载文件。输出被保存到一个临时文件。如果 wget 由于某种原因失败, 则显示输出。否则, 由于下载会在屏幕上占据较大空间, 如果命令成功则省略输出。\$state 变量随后被删除, 这样整个处理可以重新开始。

18.1.5 脚本代码



可从
wrox.com
下载源代码

```
#!/bin/bash

INFILE=${1:-${HOME}/.mozilla/firefox/*/bookmarks.html}
state=
link=
download=/tmp/download
mkdir -p "$download" 2>/dev/null
BASE_URL=http://steve-parker.org

cat $INFILE | \
    tr '\n' ' ' | tr '<' '\n' | tr '>' '\n' | tr '\t' ' ' | tr -s ' ' | \
    while read tag label
do
    if [ "$state" == "anchor" ] && [ ! -z "$tag" ]; then
        if [ "$tag" == "img" ]; then
            label=
        fi
        if [ -z "$label" ]; then
            filename=$tag
        else
            filename="$tag $label"
        fi
        origname=$filename
        suffix=1
        while [ -f "${download}/${filename}" ]
        do
            filename="${origname}.${suffix}"
            ((suffix++))
        done
        echo "Retrieving $link as $filename"
        # Prepend BASE_URL if not otherwise valid
        firstchar=`echo $link | cut -c1`
        case "$firstchar" in
            "/" ) link=${BASE_URL}$link ;;
            "#" ) link=${BASE_URL}/${link} ;;
        esac
    fi
done
```

```

esac
wget -O "${download}/${filename}" "$link" > /tmp/wget.$$ 2>&1
if [ "$?" -eq "0" ]; then
    ls -ld "${download}/${filename}"
else
    echo "Retrieving $link failed."
    cat /tmp/wget.$$
fi
state=
else
if [ "$tag" == "A" ] || [ "$tag" == "a" ]; then
    # Only do this if not already in an anchor;
    # <a href="#">a href</a> is valid!
    if [ "$state" != "anchor" ]; then
        link=`echo $label| grep -i "href=" |tr [:blank:] '\n'| \
            grep -io "href.*"|cut -c6- | tr -d '"' |tr -d "'"`
        [ ! -z "$link" ] && state=anchor
    fi
fi
fi
done
rm /tmp/wget.$$ 2>/dev/null

```

downloader.sh

18.1.6 调用结果

```

$ cat eg.html
<a href="/eg.shtml">example pages</a>
<a href="/eg2.shtml">a      pages</a>

<a      href="/e+g.shtml">eg pages</a>
<a href="/imagelink1.html"></a>
<a href="/imagelink2.html"></a>
<a bar=foo>nono</a>
This is what <a href="#">a href</a> looks like
This is what <a href="#more">a marker</a> looks like
<a href='/e_g.shtml'>more
<a href="/moreimages.html"></a>
examples
</a>
<a class="pad" href="/examples.shtml" title="eg">further examples</a>
$ ./downloader.sh eg.html
Retrieving /eg.shtml as example pages
-rw-rw-r-- 1 steve steve 10421 May 1 12:58 /tmp/download/example pages
Retrieving /eg2.shtml as a pages

```

多个空格可能造成问题。图像作为锚文本可能比较棘手。脚本使用 **img** 作为其文件名。

这一行与链接相似，但没有 **href**，所以将其忽略。

脚本会有效地关闭标签，因为它不会搜索 ****

```

-rw-rw-r-- 1 steve steve 402 May 1 12:58 /tmp/download/a pages
Retrieving /e+g.shtml as eg pages
-rw-rw-r-- 1 steve steve 2409 May 1 12:58 /tmp/download/eg pages
Retrieving /imagelink1.html as img
-rw-rw-r-- 1 steve steve 94532 May 1 12:58 /tmp/download/img
Retrieving /imagelink2.html as img.1
-rw-rw-r-- 1 steve steve 1053 May 1 12:58 /tmp/download/img.1
Retrieving # as a href
-rw-rw-r-- 1 steve steve 3407 May 1 12:58 /tmp/download/a href
Retrieving #more as a marker
-rw-rw-r-- 1 steve steve 593 May 1 12:58 /tmp/download/a marker
Retrieving /e_g.shtml as more
-rw-rw-r-- 1 steve steve 548 May 1 12:58 /tmp/download/more
Retrieving /moreimages.html as img.2
-rw-rw-r-- 1 steve steve 5930 May 1 12:58 /tmp/download/img.2
Retrieving /examples.shtml as further examples
-rw-rw-r-- 1 steve steve 50395 May 1 12:58 /tmp/download/further examples
$ ./downloader.sh eg.html ← 再次运行脚本会生成新的文件名，而不是覆盖。
$ ./downloader.sh eg.html
Retrieving /eg.shtml as example pages.1
-rw-rw-r-- 1 steve steve 10421 May 1 13:04 /tmp/download/example pages.1
Retrieving /eg2.shtml as a pages.1
-rw-rw-r-- 1 steve steve 402 May 1 13:04 /tmp/download/a pages.1
Retrieving /e+g.shtml as eg pages.1
-rw-rw-r-- 1 steve steve 2409 May 1 13:04 /tmp/download/eg pages.1
Retrieving /imagelink1.html as img.3
-rw-rw-r-- 1 steve steve 94532 May 1 13:04 /tmp/download/img.3
Retrieving /imagelink2.html as img.4
-rw-rw-r-- 1 steve steve 1053 May 1 13:04 /tmp/download/img.4
Retrieving # as a href.1
-rw-rw-r-- 1 steve steve 3407 May 1 13:04 /tmp/download/a href.1
Retrieving #more as a marker.1
-rw-rw-r-- 1 steve steve 593 May 1 13:04 /tmp/download/a marker.1
Retrieving /e_g.shtml as more.1
-rw-rw-r-- 1 steve steve 548 May 1 13:04 /tmp/download/more.1
Retrieving /moreimages.html as img.5
-rw-rw-r-- 1 steve steve 5930 May 1 13:04 /tmp/download/img.5
Retrieving /examples.shtml as further examples.1
-rw-rw-r-- 1 steve steve 50395 May 1 13:04 /tmp/download/further examples.1
$ ./downloader.sh ← 如果没有提供 HTML 文件，则使用默认的 Firefox 书签文件。
Retrieving https://addons.mozilla.org/en-US/firefox/bookmarks/ as Get
Bookmark Add-ons
-rw-rw-r-- 1 steve steve 39564 May 1 13:13 /tmp/download/Get Bookmark Add-ons
Retrieving http://www.mozilla.com/en-US/firefox/central/ as Getting Started
-rw-rw-r-- 1 steve steve 41281 May 1 13:13 /tmp/download/Getting Started
Retrieving http://fxfeeds.mozilla.com/en-US/firefox/livebookmarks/ as
Latest Headlines

```

```

-rw-rw-r-- 1 steve steve 17415 May 1 13:13 /tmp/download/Latest Headlines
Retrieving http://bad.example.com/ as This is a broken example
Retrieving http://bad.example.com/ failed.
--2011-05-01 23:03:11-- http://bad.example.com/
Resolving bad.example.com... failed: Name or service not known.
wget: unable to resolve host address `bad.example.com'
Retrieving http://www.mozilla.com/en-US/firefox/help/ as Help and Tutorials
-rw-rw-r-- 1 steve steve 25123 May 1 13:13 /tmp/download/Help and Tutorials
Retrieving http://www.mozilla.com/en-US/firefox/customize/ as Customize Firefox
-rw-rw-r-- 1 steve steve 35349 May 1 13:13 /tmp/download/Customize Firefox
Retrieving http://www.mozilla.com/en-US/firefox/community/ as Get Involved
-rw-rw-r-- 1 steve steve 5237 May 1 13:13 /tmp/download/Get Involved
Retrieving http://www.mozilla.com/en-US/firefox/about/ as About Us
-rw-rw-r-- 1 steve steve 22163 May 1 13:13 /tmp/download/About Us
$

```

18.1.7 小结

该脚本处理的输入比较复杂，且格式不严谨，所以按照文档标准来处理是不够的。用一些时间来考虑一下可能遇到的各种输入，以及如何让它们归到标准之下，这样可以使后面的处理要简单很多。

跟踪状态也很有用，尤其是在输入项是嵌套关系的时候，如注释中提到的[a href="#">a href](#)。对于其他 HTML 元素，这个状态也是必要的。case 语句可以用来轮流解析脚本支持的每个标签。

18.2 实用脚本 18-2: CSV 格式化

系统管理是一件非常吃力不讨好的事情，而且通常没有时间去设计、编写一个简洁的 shell 脚本，并对其进行测试。该脚本处理的是内层泄漏的问题。有时这种情况下的最优解决方案是创建一个非常快速与简单的脚本来提取后面要进行分析的相关数据。重要的是要获得正确的数据，而不是数据(甚至是脚本)格式多么美观。grab-meminfo.sh 就是这样的一个脚本。它只是每分钟提取 3 次时间戳与/proc/meminfo 的副本。最好是将除了需要的以外更多的数据保存起来，而不是丢掉将来会用到的一些细节。

18.2.1 用到的技术

- /proc/meminfo
- ((suffix++))
- CSV
- bc

18.2.2 概念

`/proc` 中很多文件以文本文件的形式存在,但实际上是当前运行 Linux 内核的直接接口,`/proc/meminfo` 便是其中之一。`shell` 脚本可以利用这些极好的资源以易于使用的格式得到一些内核中的原始数据,而不需要依赖于额外的实用程序(如 `free`)。这些额外的实用程序从内核中得到同样的数据,然后重新解析。

`plot-graph.sh` 脚本从 `/proc/meminfo` 中获取原始数据,然后将其格式化为 Microsoft Excel、OpenOffice.org 或 LibreOffice 这样的电子表格软件可以读取的 CSV 文件。桌面软件可以用来将数据格式化为图形,有助于对内存随时间推移的使用情况进行可视化。它还可以将数据保存为更简洁的格式。最后, `stats.sh` 对输出进行一些更长期的分析,分析的内容是较长时间里面内存用量的峰值与平均值,目的是为了找到造成问题的原因。

18.2.3 潜在的陷阱

尽管时间非常重要,但在编写像 `grab-meminfo.sh` 这样的脚本时进行一些计划还是值得的。使用脚本对输出的分析越简单越好,但尽量保存数据而不是计算总的内存使用量之后扔掉其他数据也是有好处的。通过保存完整的 `meminfo` 文件,我们可以画出内存使用量随交换分区使用量变化的图形。如果脚本只收集内存使用量的数据,则那些信息就会丢失。

如果 `grab-meminfo.sh` 创建日志文件,且格式与 `plot-graph.sh` 保存的文件相同,这样就更好了。实际上,记录下来的日志格式会不整齐,所以该脚本特意展示了如何从最初的原始状态转换为格式化的图形,而不是从经过很好地格式化之后的状态进行转换。

18.2.4 脚本结构

`grab-meminfo.sh` 脚本的代码其实没有太多组织结构,它就是一个提取 `/proc/meminfo` 内容的 `while` 循环。`plot-graph.sh` 接收数据,并将它编写成一个 CSV 文件。因为每次读取都会写入不同的文件,所以能用 `ls -tr` 利用文件的时间戳按顺序读取。如果时间戳丢失(可能是在向远程服务器复制的时候),则需要用 `sort -n` 来按照文件名排序。相关的数据用 `grep`(与 `/proc` 中的很多文件一样, `meminfo` 被设计成易于处理)从文件中提取出来,然后计算当时正在使用的物理内存与交换分区的大小,并存储在具有较好格式的日志文件中,以备将来使用。另外还向标准输出回显保存的内容。这些内容可以被重定向到 CSV 文件。

`bc` 用来将 `/proc/meminfo` 报告的千字节数转换为更容易管理的兆字节数。该脚本使用 `bc` 的 `scale` 特性。该特性在这里用于在两个小数位的精确度下进行转换。对于图形而言,这已经足够精确,同时能保证图中标注的大小是 0~18 GB,而不是 0~18 000 000 KB。

在 OpenOffice.org 中选择 B、E 与 F(Time、GB Memory Used 与 GB Swap Used)这 3 列可以生成图 18-1。在 Insert 菜单中选择 Chart,选择线状图,然后在 Chart Elements 部分添加标题。

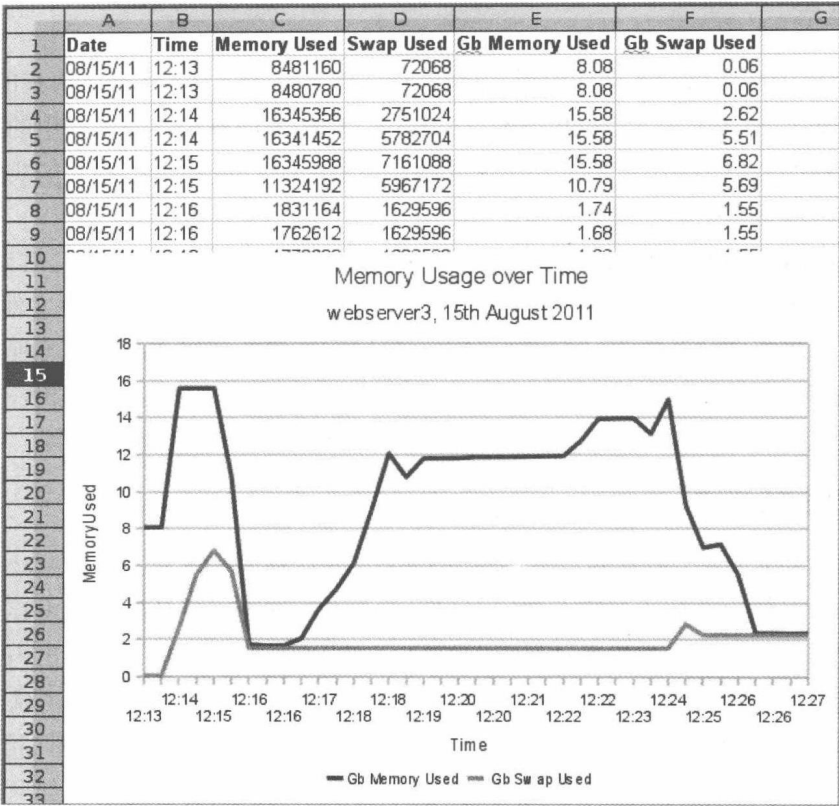


图 18-1

过一段时间之后，我们就应当看看服务器是如何使用内存的。第三个脚本 `stats.sh` 从更高的层面计算每天内存使用量的峰值与平均值。`stats.sh` 从日志文件中截取掉每个日期，然后对文件依次进行处理。如果服务器只在工作日的 9 点到 6 点使用，则平均值会出现偏移，因为统计了 24/7 的时间。这一问题可以用一条简单的 `cut` 命令来缓解。如果时间在上午 9 点之前或者下午 5 点之后，则跳过循环并忽略数据。这样保证了 `stats.sh` 只显示上午 9 点到下午 5 点 59 分之间的数据。

因为 `plot-graph.sh` 对数据的格式化比 `grab-meminfo.sh` 要好一些，所以 `stats.sh` 的任务要更轻松一些。每行内存之后都是一行交换分区，所以在 `while` 循环中再进行一次 `read` 操作来得到相关的交换分区数据。总量与峰值得到计算，且数量增加了。然后数据被保存到一个临时文件。因为 `while` 循环本身是一个子 `shell`，当循环终止后，子 `shell` 及其环境变量会消失。因此，主脚本从临时文件中读取循环的状态，计算平均值，并将数据写入到 CSV 文件。图 18-2 清晰地显示出每到周一都会有一个高峰。最上方的线表示 **Peak RAM**，然后是 **Mean RAM** 与 **Peak Swap**，**Mean Swap** 在最下方。该信息可以用来精确地找出周一有什么不同之处造成这样的问题。可能的情况是，周一是负载高峰期，服务器需要更多的内存来处理这些负载；或者是应用程序的某一部分只在周一会运行，而该部分包含需要修复的内存泄漏问题。

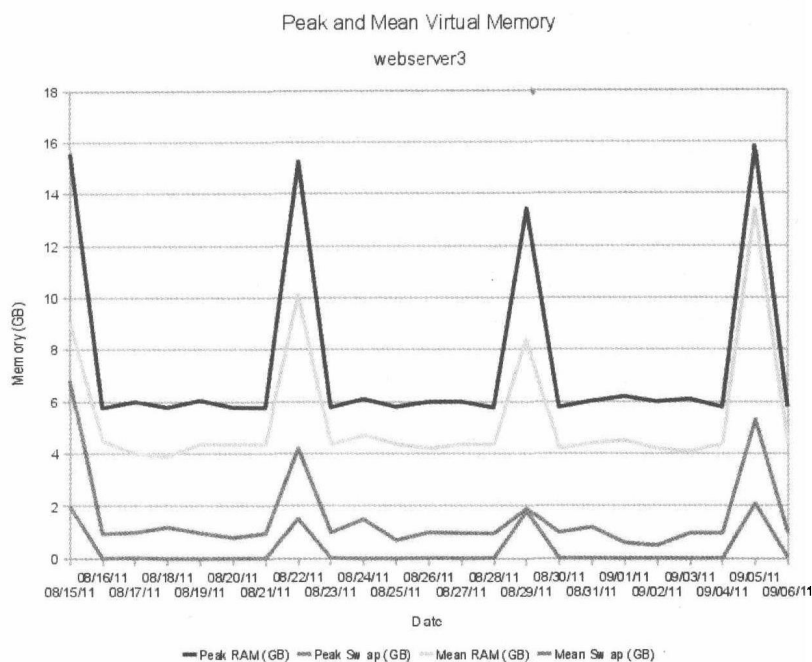


图 18-2

18.2.5 脚本代码



可从
wrox.com
下载源代码

```
#!/bin/bash
```

```
count=1
```

```
while :
```

```
do
```

```
date +%D:%H:%M > /var/tmp/$count.meminfo
```

```
cat /proc/meminfo >> /var/tmp/$count.meminfo
```

```
((count++))
```

```
sleep 20
```

```
done
```

grab-meminfo.sh

```
#!/bin/bash
```

```
LOG=/var/tmp/memory.log
```

```
echo "Date,Time,Memory Used,Swap Used,Gb Memory Used,Gb Swap Used"
```

```
for MEMINFO in `ls /var/tmp/*.meminfo | sort -n`
```

```
do
```

```
timestamp=`head -1 $MEMINFO`
```

```
memtotal=`grep "^MemTotal:" $MEMINFO | awk '{ print $2 }'`
```

```
memfree=`grep "^MemFree:" $MEMINFO | awk '{ print $2 }'`
```

```
swaptotal=`grep "^SwapTotal:" $MEMINFO | awk '{ print $2 }'`
```

```
swapfree=`grep "^SwapFree:" $MEMINFO | awk '{ print $2 }'`
```

```
ramused=$(( memtotal - memfree ))
```

```
swapused=$(( swaptotal - swapfree ))

date=`echo $timestamp | cut -d: -f1`
time=`echo $timestamp | cut -d: -f2-`

echo "$DATE Memory $ramused kB in use" >> $LOG
echo "$DATE Swap $swapused kB in use" >> $LOG

gbramused=`echo "scale=2;$ramused / 1024 / 1024"| bc`
gbswapused=`echo "scale=2;$swapused / 1024 / 1024"| bc`

echo "$date,$time,$ramused,$swapused,$gbramused,$gbswapused"
done
```

plot-graph.sh

```
#!/bin/bash
LOG=${1:-memory.log}
CSV=${2:-stats.csv}

echo "Date,Peak RAM,Peak Swap,Mean RAM,Mean Swap,Peak RAM (GB),\
Peak Swap (GB),Mean RAM (GB),Mean Swap (GB)" > $CSV

totals=/tmp/total.$$

for date in `cat $LOG | cut -d":" -f1 | sort -u`
do
    count=0
    peakram=0
    peakswap=0
    totalram=0
    totalswap=0
    echo "Processing $date"
    grep "^${date}:" $LOG | while read timestamp type ramused text
    do
        hour=`echo $timestamp|cut -d: -f2`
        if [ "$hour" -lt "9" ] || [ "$hour" -gt "17" ]; then
            continue
        fi
        read timestamp swaptype swapused text text
        ((count++))
        echo count=$count > $counter
        let totalram=$totalram+$ramused
        let totalswap=$totalswap+$swapused
        [ $ramused -gt $peakram ] && peakram=$ramused
        [ $swapused -gt $peakswap ] && peakswap=$swapused
        echo totalram=$totalram > $totals
        echo totalswap=$totalswap >> $totals
        echo peakram=$peakram >> $totals
        echo peakswap=$peakswap >> $totals
    done
done
```

```

    echo count=$count >> $totals
done
. $totals
meanram=`echo "$totalram / $count" | bc`
meanswap=`echo "$totalswap / $count" | bc`

peakramgb=`echo "scale=2;$peakram / 1024 / 1024"| bc`
peakswapgb=`echo "scale=2;$peakswap / 1024 / 1024"| bc`
meanramgb=`echo "scale=2;$meanram / 1024 / 1024"| bc`
meanswapgb=`echo "scale=2;$meanswap / 1024 / 1024"| bc`

echo "$date,$peakram,$peakswap,$meanram,$meanswap,$peakramgb,$peakswapgb,\
$meanramgb,$meanswapgb" >> $CSV
done
rm -f $totals

```

stats.sh

18.2.6 调用结果

```

$ nohup ./grab-meminfo.sh &
[1] 18580
$ nohup: ignoring input and appending output to `nohup.out'

$ ./plot-graph.sh > memory.csv
$ oocalc memory.csv

$ ./stats.sh
Processing 08/15/11
Processing 08/16/11
Processing 08/17/11
Processing 08/18/11
Processing 08/19/11
Processing 08/20/11
Processing 08/21/11
Processing 08/22/11
Processing 08/23/11
Processing 08/24/11
Processing 08/25/11
Processing 08/26/11
Processing 08/27/11
Processing 08/28/11
Processing 08/29/11
Processing 08/30/11
Processing 08/31/11
Processing 09/01/11
Processing 09/02/11
Processing 09/03/11
Processing 09/04/11
Processing 09/05/11
Processing 09/06/11
$ oocalc stats.csv

```

18.2.7 小结

shell 是非常有用的收集与处理数据的工具，但对于正式的演示而言，纯文本的接口会受到限制。让 shell 向桌面应用程序输出数据，这样的应用程序有 Web 浏览器(如第 15 章的实用脚本 15-2 介绍的)以及电子表格(上面的脚本)。它们分别使用 HTML 与 CSV 这样的基于文本的文件格式。这样便可以轻松地完成从文本到图形化的转换。这样的数据处理在其他环境中是无法实现的。如果数据一开始便是 Excel 电子表格与宏的形式，则会总是局限于 Excel。但如果使用常规的基于文本的格式，则可以通过适当的工具对数据进行重新组织与格式化，并在必要的时候输出到更加封闭的应用程序中。即使是与 Excel 这样的大型软件项目比起来，不起眼的 shell 脚本也可以更加强大与灵活。

第 19 章

数 值

数值是计算的中心，但本章介绍的是在 shell 脚本中处理数值时可能遇到的一些问题。第一个实用脚本用 3 种方法列出斐波那契数列。它揭示了一些 shell 能处理的数值大小的局限，并介绍了一些解除这些局限的方法。

第二个实用脚本对数值在不同进制之间转换。尽管通常用十进制表示数值，但 CPU 使用的原生格式是二进制。十六进制也很常用，因为它比二进制更加紧凑，而且一个字节可以整齐地表示为两个十六进制字符。netboot.sh 脚本使用 printf 将数值在十进制与十六进制之间进行转换。bc 工具可以将数值从任意进制向其他进制转换，但 printf 提供了最简单的以十六进制与八进制表示十进制数的方法。

19.1 实用脚本 19-1：斐波那契数列

斐波那契数列是非常简单的整数数列，其中数列中的每个数都由前两个数的和计算得到。通常是从 0 与 1 开始，所以 $F(0)$ 为 0， $F(1)$ 为 1。而 $F(2)$ 为 $F(0)+F(1)$ ，等于 1。 $F(3)$ 为 $F(1)+F(2)$ ，等于 2。 $F(4)$ 为 3， $F(5)$ 为 5， $F(6)$ 为 8， $F(7)$ 为 13， $F(8)$ 为 21。因为该数列以非常快的速度增长，所以这里我们不是考虑其美学上的意义，而是来看我们使用的 3 种方法如何用 shell 来处理较大的数值。

19.1.1 用到的技术

- 函数
- `((count++))`
- `$((x + y))`
- `[x -lt y]`
- `expr`
- `bc`

19.1.2 概念

不断将两个数相加的思想非常简单，但这种思想是非常普遍的。图 19-1 显示螺旋线可以用斐波那契数列的形式表示。每个粗线表示的格子的边长都等于数列中其位置上的数，所以前两个格子为 1×1，然后是 2×2，随后是 3×3、5×5、8×8、13×13 与 21×21 等。在每个粗线表示的格子中画 1/4 圆就可以得到螺旋线。螺旋线在自然界中很常见，如菠萝、蜗牛壳、向日葵。

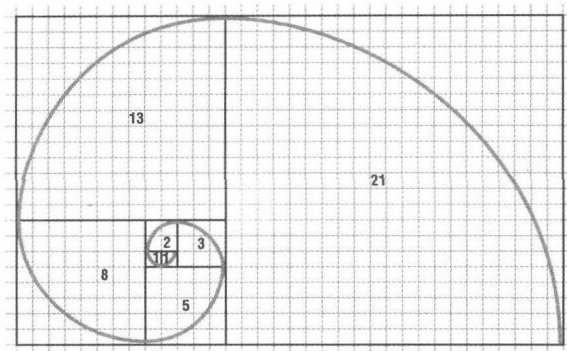


图 19-1

19.1.3 潜在的陷阱

处理较大数值时的主要陷阱在于，存储会自己溢出(回绕)。计算机内部使用二进制，其中一个比特表示开启(1)或关闭(0)。8 比特的单字节可以最大表示到 255。超过 255 的话，数值又会从 0 开始。

在一般的十进制数学中，9 999 之后的数是 10 000，然后是 10 001 与 10 002 等。如果有一个只显示最低位的 4 个数字的计算器，则 9999 之后的数看起来就是 0000，然后是 0001 与 0002。同样的情况也发生在对数值的计算机内部表示中。1111 1111(255)之后是 1 0000 0000(256)，但 8 比特的字节只会保存 8 个最低位，它们全部为 0。

表 19-1 位溢出

二进制数值	十进制数值
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
0000 0100	4
0000 0101	5
0000 0110	6
0000 0111	7
1111 1100	252

(续表)

二进制数值	十进制数值
1111 1101	253
1111 1110	254
1111 1111	255
0000 0001 0000 0000	256 或 0
0000 0001 0000 0001	257 或 1
0000 0001 0000 0010	258 或 2

方法一的脚本使用单字节精度，并且发生了上述问题。随后的脚本使用 4 字节组成的一个字来表示数值，但也会出现这样的问题。一个字能表示的最大数由 32 个连续的 1 组成，即 4 294 967 295。超过这个数后，32 位的存储空间还是会溢出到 0。尽管用 64 位最大能表示 18 446 744 073 709 551 615，如方法二的脚本所示，但这么大的范围不一定是可用的。


19.1.4 方法一的结构

第一个脚本与其他脚本在结构上稍有不同，它使用 `seq` 在 `for` 循环中计数。因为脚本很快会失败，所以用 `seq` 来控制很有用，并且能保证很快停止。之后，更健壮的版本使用 `while` 无限循环(`while` 中的分号总为真)，这样脚本会一直运行直到较大的数值后失败。

第一个脚本使用 `fibonacci` 函数。该函数返回两个参数的和。在计算机科学中，这是传统的函数用法。这种用法接收数值作为输入，然后返回单个数值作为计算的结果。不过，返回码本来是用于状态报告，而不是用来表示大整数的。所以在处理超过 255 的数时，这种方法在 `shell` 中没有用处。脚本包含了一个基本的合理性检查，即是否发生溢出。而且数值总是会变大。 $144+233=377$ ，二进制表示就是 1 0111 1001。扔掉第 9 位就剩下 0111 1001，十进制表示为 121。121 比前一个数 233 小，所以答案 121 显然是错误的。肯定有更好的办法，而随后的一些方法则使用了另外一些技术。

每次调用一个 0.1 秒的休眠，这样更容易看清输出内容；否则，输出的速度会大于我们阅读的速度。当然，这种人工延迟可以按照需求进行一些修改或删除掉。这里的 `fibonacci` 函数会运行 15 次，所以没有太大的影响，但如果后面是成千上万次的迭代就大有不同了。第 17 章提到过，不是所有的 `sleep` 实现都能执行亚秒级的延迟。这种情况下只能延迟整整一秒或完全不延迟。

19.1.5 方法一的脚本



```
#!/bin/bash

function fibonacci
{
    return $(( $1 + $2 ))
}

F0=0
```

可从
wrox.com
下载源代码


```
F1=1
echo "0: $F0,"
echo "1: $F1, "
for count in `seq 2 17`
do
    fibonacci $F0 $F1
    F2=$?
    if [ "$F2" -lt "$F1" ]; then
        echo "${count}: $F2 (WRONG!), "
    else
        echo "${count}: $F2,"
    fi
    F0=$F1
    F1=$F2
    sleep 0.1
done
fibonacci $F0 $F1
echo "${count}: $?"
```

fibonacci1.sh

19.1.6 方法一的调用结果

```
$ ./fibonacci1.sh
0: 0,
1: 1,
2: 1,
3: 2,
4: 3,
5: 5,
6: 8,
7: 13,
8: 21,
9: 34,
10: 55,
11: 89,
12: 144,
13: 233,
14: 121 (WRONG!),
15: 98 (WRONG!),
16: 219,
17: 61 (WRONG!),
17: 24
$
```

19.1.7 方法二的结构

第二个斐波那契脚本忽略函数的返回码，而直接让 `fibonacci` 函数将结果回显出来。`shell` 本身可以处理超过 255 的整数；只有返回码才限制为 1 字节。这样可以绕过 1 字节的限制，且调用函数的代码定义了 `F2=`fibonacci $F0 $F1``，于是 `F2` 获得函数的标准输出，而

不是返回码。

该脚本还用 `while` 无限循环替换掉了具有预定义终止状态的 `for` 循环。脚本在 `while` 循环中使用 `count` 变量，并增加其数值，因为有了 `for` 循环。该变量用 `bash` 语法 `((count++))` 来增加，这与 C 语句 `count++` 非常相似。它是 `let count=count+1` 的简写。前者效率并不会更高，只是易于编写与阅读。

该脚本算到了数列中的第 92 个数，即 7 540 113 804 746 346 429，大概是 7.5 百万的三次方。第 93 个数是 4 660 046 610 375 530 309+7 540 113 804 746 346 429，结果为 12 200 160 415 121 876 738，大概是 12.2 百万的三次方。这就是 `shell` 处理能力的极限了。`shell` 的整数实现调用二进制补码系统来表示正负整数。以损失一部分表示范围为代价，变量就可以存储负数与正数。1 个字节(8 位)的二进制补码可以表示 -127~128 之间的整数，而不是 0~255。64 位的二进制补码可以表示 -9 223 372 036 854 775 808 ~ +9 223 372 036 854 775 807 之间的整数。因此，该脚本在数列的第 92 个数之后失败，因为第 93 个数大约是 12 百万的三次方，大于最大能表示的 9.2 百万的三次方。

用 `expr` 代替 `shell` 的数学计算也没有用，因为问题在于数值的内部表示，而不是实现加法的技术。`fibonacci3.sh` 将 `fibonacci` 函数中的 `echo` 替换为 `expr`，但与 `fibonacci2.sh` 失败的原因几乎相同。唯一的区别就是 `expr` 会给出一条错误消息，而 `shell` 实现会静默失败。因为 `expr` 没有给出数值，所以测试 `if ["$F2" -lt "$F1"]` 会扩展为 `if ["" -lt "$F1"]`。这在语义上不成立，所以 `test`(又称 `[]`)会报告 `integer expression expected`。

19.1.8 方法二的脚本



可从
wrox.com
下载源代码

```
#!/bin/bash

function fibonacci
{
    echo $(( $1 + $2 ))
}

F0=0
F1=1
echo "0: $F0, "
echo "1: $F1, "
count=2
while :
do
    F2=`fibonacci $F0 $F1`
    if [ "$F2" -lt "$F1" ]; then
        echo "${count}: $F2 (WRONG!),"
    else
        echo "${count}: $F2,"
    fi
    ((count++))
    F0=$F1
    F1=$F2
    sleep 0.1
done
```

```
done
fibonacci $F0 $F1
```

fibonacci2.sh

```
#!/bin/bash

function fibonacci
{
    #echo $(( $1 + $2 ))
    expr $1 + $2
}

F0=0
F1=1
echo "0: $F0, "
echo "1: $F1, "
count=2
while :
do
    F2=`fibonacci $F0 $F1`
    if [ "$F2" -lt "$F1" ]; then
        echo "${count}: $F2 (WRONG!),"
    else
        echo "${count}: $F2,"
    fi
    ((count++))
    F0=$F1
    F1=$F2
    sleep 0.1
done
fibonacci $F0 $F1
```

fibonacci3.sh

19.1.9 方法二的调用结果

```
$ ./fibonacci2.sh
0: 0,
1: 1,
2: 1,
3: 2,
4: 3,
5: 5,
6: 8,
7: 13,
( 81 lines of output omitted )
89: 1779979416004714189,
90: 2880067194370816120,
91: 4660046610375530309,
```

```

92: 7540113804746346429,
93: -6246583658587674878 (WRONG!),
94: 1293530146158671551,
95: -4953053512429003327 (WRONG!),
96: -3659523366270331776,
97: -8612576878699335103 (WRONG!),
98: 6174643828739884737,
99: -2437933049959450366 (WRONG!),
^C
$ ./fibonacci3.sh
0: 0,
1: 1,
2: 1,
3: 2,
4: 3,
5: 5,
6: 8,
7: 13,
( 81 lines of output omitted )
89: 1779979416004714189,
90: 2880067194370816120,
91: 4660046610375530309,
92: 7540113804746346429,
expr: +: Numerical result out of range
./fibonacci3.sh: line 17: [: : integer expression expected
93: ,
expr: syntax error
./fibonacci3.sh: line 17: [: : integer expression expected
94: ,
expr: syntax error
./fibonacci3.sh: line 17: [: : integer expression expected
95: ,
expr: syntax error
./fibonacci3.sh: line 17: [: : integer expression expected
96: ,
expr: syntax error
./fibonacci3.sh: line 17: [: : integer expression expected
97: ,
^C

```

19.1.10 方法三的结构

第四个脚本使用 `bc` 进行数学计算, 其中起作用的是 `shell` 与变量类型之间的怪异关系。尽管 `shell` 不能处理大于 9 223 372 036 854 775 807 的整数, 但可以处理看起来像整数的字符串。因为 `shell` 不进行数学计算, 所以它能执行得比之前的脚本更远。然而, 比较测试

["\$F2" -lt "\$F1"]确实会将这些大数值当成整数处理，且会像 `fibonacci3.sh` 那样在数列的第 92 个数之后给出错误消息。为了美观起见，`fibonacci4.sh` 将 `test` 移除。该脚本能计算到第 324 个数。出于格式化的目的，`bc` 在每第 69 个数字之后都会输出一个反斜线(\)与一个换行符(\n)。这是标准的续行表示法。向 `bc` 调用追加 `| tr -d '\\\n'` 会将反斜线与换行符删除。不过下面也显示了没有续行的输出。输出在第 4374 行之后还在继续，但之后会变得太长而难以显示在书本中。

19.1.11 方法三的本脚本



可从
wrox.com
下载源代码

```
#!/bin/bash

function fibonacci
{
    echo $1 + $2 | bc | tr -d '\\\n'
}

F0=0
F1=1
echo "0: $F0, "
echo "1: $F1, "
count=2
while :
do
    F2=`fibonacci $F0 $F1`
    echo "${count}: $F2,"
    ((count++))
    F0=$F1
    F1=$F2
    sleep 0.1
done
fibonacci $F0 $F1
```

fibonacci4.sh

19.1.12 方法三的调用结果

```
$ ./fibonacci4-without-tr.sh
0: 0,
1: 1,
2: 1,
3: 2,
4: 3,
5: 5,
6: 8,
7: 13,
(316 lines omitted)
324: 23041483585524168262220906489642018075101617466780496790573690289968,
325: 37281903592600898879479448409585328515842582885579275203077366912825,
326: 60323387178125067141700354899227346590944200352359771993651057202793,
327: 97605290770725966021179803308812675106786783237939047196728424115618,
```

```

328: 15792867794885103316288015820804002169773098359029881919037948131841\
1,
(standard_in) 1: syntax error
329: ,
(standard_in) 1: illegal character: \
^C
$ ./fibonacci4.sh
0: 0,
1: 1,
2: 1,
3: 2,
4: 3,
5: 5,
6: 8,
7: 13,
(4364 lines omitted)
4372: 22104371486889933602618735961044632349577043348049549237830063379958351358033
11268917254596149645711206174771088398441721789053643030921941304454073026431478192
41390892330235850453528642957368751545308642998939685610815800594399228263960769129
21254068071415653174058777017937238760614372901733356868545935069969579993502736008
08716613231538450869710003373518827779200816776218994622421477019847855133314714546
18899014113986652857034659252548051809757752694982390345887265182046582236328217001
33763823103115778273585794003560474902759766584168221196134268569815826322073363147
42869925350910884706399169984309965059260700352426234305362961676292134793147315180
46138610752348150619982159457883646380864526446597385913849760882251106843955544650
97167798761562880904242190144594305156929701780458050593218239235282940068118503196
77599480281086457168264167341641925748865316857196662281426267110666547911046191378
6584739,
4373: 35765624365741963284919524730378813338126540562607632483122508004471265309976
21921732591032431764684384346503043840940743739673231228573220536605893452946866910
24663796720201105157982056926986064130753541939199360054474934714408493656406783979
93916754837161602464272387025650369909799845370879766002550803144584837747973615962
90818951240614280213820290475677516870911282615410106043042221375461525790866579223
58682302741351708415973899505187084097555426806237660513829820443468851292352812652
56208788452517468884050138332815587507021627661962415808212786797557853665691000413
38289519843703519722711363485632557273400445192109609864836613146184139760272144933
24088664108431699850825644193287950483122088101345491185485939837331355185435692969
42157189659550021594510239281601412130301689135398452673470238178668881272347228053
08054539175807683506731517912644386761586494590133000407819344010943092478942749225
2486693,
4374: 57869995852631896887538260691423445687703583910657181720952571384429616668009
33190649845628581410395590521274132239382465528726874259495161841059966479378345102
66054689050436955611510699884354815676062184938139045665290735308807721920367553109
15170822908577255638331164043587608670414218272613122871096738214554417741476351970
99535564472152731083530293849196344650112099391629100665463698395309380924181293769
77581316855338361273008558757735135907313179501220050859717085625515433528681029653
89972611555633247157635932336376062409781394246130637004347055367373679987764363560
81159445194614404429110533469942522332661145544535844170199574822476274553419460113

```

数值的宽度决定了最后的 1 要在反斜线与换行符之后单独放置在一行。shell 以单个字符串 (因为这时 shell 将输出当成字符串而不是数值) 读取时必须将反斜线与换行符删除。

```
70227274860779850470807803651171596863986614547942877099335700719582462029391237620
39324988421112902498752429426195717287231390915856503266688477413951821340465731249
85654019456894140674995685254286312510451811447329662689245611121609640389988940603
9071432
```

19.1.13 小结

从函数中获取结果的方法各式各样。返回码只有单个字节，且总是与 Unix 标准兼容。这使得这种方法在向调用者返回值时不是那么理想。然而，结果可以写向标准输出或者要求的任何其他文件。这种方法的缺点是绝不可以产生其他输出，因为所有的输出都会被当成计算的结果来解析。

shell 可以进行较为基础的数学计算，但对于非常大的数值，则必须使用像 `bc` 这样专门的工具。如果要完成很多复杂的功能，因为 `bc` 本身是可编程的，所以 `bc` 脚本比 `shell` 脚本用处更大。`bc` 的数学库还包含了三角几何与其他更高级的功能。

19.2 实用脚本 19-2: PXE 启动

过去经常会遇到这样的情况：服务器要从磁带、软盘、CD 或 DVD 启动来安装，而且安装人员要回答一系列问题使操作系统按照要求进行安装。在今天，这样的情况也会发生，但大型机构都没有时间为它们要安装的成千台服务器执行这样低层的重复性操作。小型机构也会受益于 PXE 这种方法带来的快速、自动化以及相同的安装过程。这样就需要自动化且无人值守的网络安装。而且 DHCP 与 PXE 在很大程度上是在 x86 架构下实现这种安装的唯一方式。RedHat 的 Kickstart 系统是 Linux 下使用最广泛的自动化安装架构。该脚本提供了用于执行网络安装的最简单的基础性配置。

因为本书并不是介绍系统管理，所以这里不介绍如何配置 DHCP、TFTP 与 NFS 服务器。本书可以作为深入这些话题的一个起点，但真正的目的是为了演示如何在 shell 脚本中处理像 IP 地址这样的数，甚至不使用明显的数学计算。`printf` 命令可以随时进行很多转换操作。

19.2.1 用到的技术

- PXE
- Kickstart
- `printf`

19.2.2 概念

当计算机使用 PXE(预执行环境)直接从网络启动的时候，它需要从 DHCP 服务器获取一个 IP 地址。DHCP 服务器可以给该计算机提供 TFTP 服务器的详细信息，从中可以获取到一个可执行文件。一般对于 Linux 客户端，该文件名为 `/linux-install/pxelinux.0`。一旦客户端获取并执行 `pxelinux.0`，它便会经过一次硬编码，目的是从 `pxelinux.0` 所在目录的子目

录 `pxelinux.cfg/` 中寻找一个文件。首先，它会寻找以 MAC 地址命名的文件，形式为 `01-xx-xx-xx-xx-xx-xx`。然后，再寻找以 DHCP 服务器提供的 IP 地址命名的文件。

IP 地址是以十六进制格式查找的。即如果用十六进制表示，192 是 `0xC0`，168 是 `0xA8`，1 是 `0x01`，42 是 `0x2A`，所以 192.168.1.42 就是 `0xC0A8012A`。该脚本设置了一个非常基本的安装环境，并使用 `printf` 的格式化功能以十六进制格式显示 IP 地址。脚本没有执行任何特别繁重的计算任务。

19.2.3 潜在的陷阱

除了安装错误的服务器或者使用了错误的配置这样宽泛的问题，没有什么特别的陷阱。`bc` 也能用来进行进制转换，并且是任意进制。`printf` 则局限于八进制或十六进制输出。192.168.1.42 可以转换为相等的十六进制形式 `0xC0A8012A`，但比脚本使用的 `printf` 方案进行了更深入的解析。

```
$ IP=192.168.1.42
$ echo "obase=16;$IP" | tr '.' '\n' | bc
C0
A8
1
2A
$
```

19.2.4 脚本结构

关于数值操作，脚本中的主要部分是下面这一行：

```
CLIENT_HEXADDR=$(printf "%02X%02X%02X%02X" `echo $CLIENT_IP | tr '.' ' '`)
```

该命令会回显 `$CLIENT_IP` 变量(以 192.168.1.42 这样的格式)，然后将点号转换为空格。也就是说 192、68、1 与 42 成了 `printf` 命令的独立参数。`printf` 调用使用的格式化字符串是 `%02X%02X%02X%02X`，它将输入的 4 个数转换为两字符的大写十六进制格式，不够大就填零。这就是 `pxelinux.0` 要寻找的文件格式。

`%x` 将任何十进制数转换为相等的十六进制数，大于 9 的数用小写 `a~f` 表示。`%X` 也是转换为十六进制，但使用大写 `A~F`。与简单的 `%X` 相比，`%02X` 表示数值会通过填零以保证至少两个字符的宽度。这确保了 1 被转换为 01，而不是单个 1。`C0A8012A` 与 `C0A812A` 不是同一个文件名，且 `pxelinux.0` 读取每个 IP 地址中的八位位组作为文件名的单个字节。

```
$ printf "%x%x%x%x\n" 192 168 1 42
c0a812a
$ printf "%02x%02x%02x%02x\n" 192 168 1 42
c0a8012a
$ printf "%02X%02X%02X%02X\n" 192 168 1 42
C0A8012A
$
```


脚本的余下部分通过\$TFTPBOOT/messages/中的模板文件为安装创建必要的文件。`create_msgs` 函数只为客户端创建一对菜单文件,其中包含了显示出的客户端与服务器的名称。`create_kickstart` 创建一个非常简短的 Kickstart 文件,可以让 RedHat 安装程序用来对安装进行配置。实际中的 Kickstart 文件要更长,并且可以包含磁盘布局的要求、`postinstall` 脚本,以及为了完全开启无人值守安装所要安装与不安装的软件包列表。如果要对安装进行调整,可以在`%post` 部分添加代码。添加的代码可能相当简短,但安装之后它会启动对客户终端进行自定义的全部脚本。该脚本的`%post` 部分向`/etc/hosts` 中添加了一项时间服务器,然后还调用了 NFS 服务器上针对客户端的脚本(如果有的话),目的是执行针对客户端的 `postinstall` 任务。这可以作为一些可能需要的繁重的自定义任务的起始阶段。

`create_pxelinux_file` 创建的配置文件被传递到 `pxelinux.0` 来显示一个基本菜单。当用户按下 F2 键时,菜单会给出一些额外的文本(来自 `client-f2.txt`)。在启动或按下 F1 键时会给出基本文本(来自 `client.txt`)。`messages/` 目录在 `$TFTPBOOT` 目录中。类似地, `${OSNAME}/vmlinuz` 与 `${OSNAME}/initrd.img` 文件则分别指向 TFTP 服务器上的 `/tftpboot/RHEL60/vmlinuz` 与 `/tftpboot/RHEL60/initrd.img`。

该脚本最主要的函数是 `calc_client_details`。在脚本的开头, `$CLIENT` 变量被赋值为 ``getent hosts $1`` 输出的第一个字段(主机名)。无论传递给脚本的是主机名还是 IP 地址,都能得到主机名。因为无论传递给 `getent` 的是哪个值, `getent hosts` 总是以相同的格式返回数据。然后 `calc_client_details` 查询该主机名,并获得第二个字段,即 IP 地址。函数接着按照前面描述的那样处理 IP 地址。如果结果是一个八字符的字符串,则假设已经成功找到主机名与 IP 地址,并且将 IP 地址转换为可用的十六进制字符串。如果很显然成功了,则函数在一行内显示所有这些信息,然后继续执行。如果没有成功,则显示计算结果,并在进行任何写操作之前退出。

19.2.5 脚本代码



可从
wrox.com
下载源代码

```
#!/bin/bash

TFTPBOOT=/tftpboot/linux-install/pxelinux.cfg
NFS=/kickstart
CLIENT=`getent hosts $1 | awk '{ print $2 }'`
if [ -z "$CLIENT" ]; then
    echo "A failure occurred in looking up \"$1\""
    exit 2
fi
SERVER=`hostname`
OSNAME=RHEL60

function calc_client_details
{
    CLIENT_IP=`getent hosts $CLIENT | awk '{ print $1 }'`
    if [ -z "$CLIENT_IP" ] || [ -z "$CLIENT" ]; then
        echo "A failure occurred in looking up \"$CLIENT\""
        exit 2
    fi
}
```


declan will be installed over the network, destroying the operating system currently installed on the internal disk.

This will be installed from goldie. If this is not what you want, type boot to boot from the internal disks.

Press F1 for the main install screen
Press F2 for this screen

```
goldie# cat /tftpboot/linux-install/pxelinux.cfg/0A8010A
default boot
timeout 600
prompt 1
display messages/declan.txt
F1 messages/declan.txt
F2 messages/declan-F2.txt

label boot
    localboot 0
label install
    kernel RHEL60/vmlinuz
    append initrd=RHEL60/initrd.img ks=nfs:goldie:/kickstart/declan.cfg
goldie# cat /kickstart/declan.cfg
# Kickstart file for declan to boot from goldie
text install
# You would probably want to put more details here
# but this is a shell scripting recipe not a kickstart recipe
goldie#
```

一旦 declan 从网络中启动，则会在图 19-2 中显示出来。图中显示了菜单，并且在管理员输入 install 时，从 TFTP 服务器加载 vmLinux，然后加载 initrd.img。

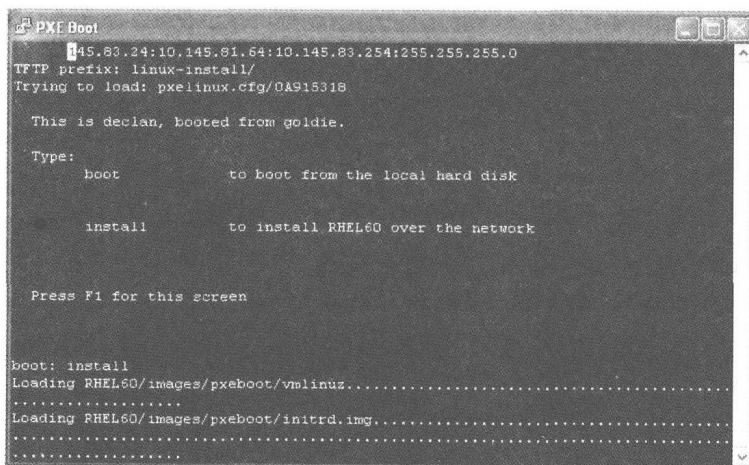


图 19-2

CLIENT_NAME_HERE will be installed over the network, destroying
the operating system currently installed on the internal disk.

This will be installed from SERVER_NAME_HERE. If this is not what you
want, type boot to boot from the internal disks.

Press F1 for the main install screen
Press F2 for this screen

client-f2.txt

19.2.6 调用结果

```
goldie# ./netboot.sh delan      delan 这个主机名无法解析, 应当纠正为 declan.
A failure occurred in looking up "delan" ←
goldie# ./netboot.sh declan
Client details: declan is at IP address 192.168.1.10 (COA8010A)
-rw-r--r-- 1 root root 219 Apr 26 12:21 /tftpboot/linux-install/pxelinux.cfg/messages/declan.txt
-rw-r--r-- 1 root root 419 Apr 26 12:21 /tftpboot/linux-install/pxelinux.cfg/messages/declan-f2.txt
-rw-r--r-- 1 root root 174 Apr 26 12:21 /kickstart/declan.cfg
-rw-rw-r-- 1 root root 245 Apr 26 12:21 /tftpboot/linux-install/pxelinux.cfg/COA8010A
goldie# cat /tftpboot/linux-install/pxelinux.cfg/messages/declan.txt
```

This is declan, booted from goldie.

Type:

boot to boot from the local hard disk

install to install RHEL60 over the network

Press F1 for this screen
Press F2 for information on the install process

```
goldie# cat /tftpboot/linux-install/pxelinux.cfg/messages/declan-f2.txt
```

This is declan, booted from goldie.

This page provides information about the boot process.

19.2.7 小结

PXE 启动配置可能有些麻烦，但是一旦这样的架构准备就位，便是对几十、上百甚至成千个服务器进行快速、简单的自动化安装的极佳方式。该脚本介绍的数学运算涉及将一种命名方式(192.168.1.42)转换为另一种(C0A8012A)。这是有必要的，因为大多数使用 IPv4 地址的系统使用十进制表示，但 `pxelinux.0` 使用十六进制。无论哪种系统都可以，但是两种格式之间的转换是必须的。`shell` 脚本的作用是适应这些不同的特性，并在这些类似但不同的系统之间起到粘合的作用。

第 20 章

进 程

进程控制是操作系统内核的一项关键任务。内核还提供了发送信号让一个进程向另一个进程发送消息的功能。接收信号的 shell 可以用 `trap` 功能来处理这些信号。还有其他一些方法：例如，一个文件的存在性可以用来改变脚本的行为。本章介绍的脚本使用了这两种方法。该脚本还使用了 `pgrep` 与 `kill` 命令来查询运行进程以及向它们发送信号。

20.1 实用脚本 20-1：进程控制

现在的一些商业群集产品提供了对网络和存储服务以及进程的监控和故障切换功能。另外还有对多节点群集的故障场景的高级保护措施。这些故障场景通常被称为 `split brain` 与 `amnesia`。该脚本不至于那么复杂，但还是提供了简单的对单个服务器上的服务的监控与重启功能。

20.1.1 用到的技术

- 关联数组
- 信号处理
- 配置文件
- 进程控制： `pgrep`、`kill`
- `logger`
- 循环： `for`、`while`
- 条件执行： `if`、`case`、`[expression] &&`

20.1.2 概念

群集与高可用性是两个独立的大型话题。该脚本只包含了对进程的监控与可能的重启。这样的任务可能起初看来比较琐碎，但处理起来还是有一些微妙之处。例如，如何处理一个总是失败的进程。脚本会注意服务的最后失败时间，且如果之前被重启过，则禁用

该服务并不再重启。当然，仅仅因为某个服务在两周之前失效过并且又出了问题，这并不能成为放弃该服务的理由，所以脚本中定义了一个 3 分钟(180 秒)的超时。在读取配置文件之前对脚本中这样的数值与调试值进行硬编码，这样一来，如果没有其他数值，则进行默认设置，但用户可以编辑/etc/ha/ha.cfg 对这些值进行修改。

脚本的基本原则是定期检测被监控进程的 PID。如果进程已经被停止，则重启。如果已经用另一个 PID 重启，则将其作为失败处理，因此将其记录到日志且让进程继续进行。这意味着进程不会再次重启失败，但脚本没有进行直接干预。

不同的应用程序具有不同的性质。对于一些长时间运行的进程(这里使用 `sleep 600` 来表示一个最终会失败的长时间运行的进程)很简单：启动进程，监控 PID 是否为活动状态。如果进程由于某种原因死亡，则启动一个副本。有些服务并没有这么简单。Apache Web 服务器通过 `apachectl start` 指令启动，但该进程本身几乎立刻终止，留下一组 `apache2`(名称可能为 `apache`、`httpd` 或 `httpd2`，取决于具体的构建方式)进程继续运行。对于这种情况，该脚本进行一次短时间的等待，然后看都有一些什么 PID。Apache 在这里实际的工作是留下一个属于 `root` 的进程。该进程绑定到 80 特权端口，并且有自己的子进程。这些子进程以非特权用户的身份运行，且作为对收到请求进行服务的实际进程。

每个应用程序的参数由针对要监控的每个服务的.conf 文件进行配置。其中包含了一些没有使用的变量，表示还可以完成其他一些功能。除了这些未使用的变量外，这些文件中的主要变量用来表示服务是否开启、启动命令、在进程树中要查找的进程的名称，以及 PID 被收集与监控之前的延迟。另外还有一个表示进程在多次失败后是否应当被停止的标志变量。我们注意到，启动命令与最终运行且被监控的进程的名称可以完全不相同，也可以相同(如这里的 `sleep`)。

最后，脚本监控一个叫做 `Friar Tuck` 的特殊守护进程。在 <http://www.catb.org/jargon/html/meaning-of-hack.html> 中的黑客词典(Jargon File)给出了 `Robin Hood` 与 `Friar Tuck` 两个进程的有关典故。这两个进程相互监控，如果一个被关闭，则另一个会将其重启。Windows NT 3.5 与 4.0 也使用了类似的功能防止用户修改注册表来将工作站升级为服务器。一个线程监控注册表设置，另一个监控监控器。这里也使用了相同的原理，如果 `HA Monitor` 脚本出现问题，则整个系统都失效。这称为单点故障(SPoF)。单点故障可以通过使用协作进程来避免。协作进程可以监控并重启主监控器脚本。这也会使协作进程出现单点故障的情况，所以主监控器脚本要监控其协作进程。这样一来，它们就能保证对方在任何时间都在运行。对 `Friar Tuck` 进程而言，它的代码可能与主监控器脚本的完全一样，但 `Friar Tuck` 不必像主 `HA Monitor` 脚本那样复杂。也就是说，`Friar Tuck` 可以用来描述监控过程的基本操作，而主 `HA Monitor` 脚本可以更进一步用来为进程监控提供一些更通用的代码。

`Friar Tuck` 还可以作为系统的通信点。我们可以调用 `friartuck.sh` 启动该进程，然后向它发送停止服务的信号，并强制其刷新配置。否则，很难完全关闭整个框架。两个进程必须在很短的时间之内(在一个进程恢复另一个之前)关闭。这就是黑客词典中关于 `Robin Hood` 与 `Friar Tuck` 的解释。向 `Friar Tuck` 发送信号也就是让主脚本即时对配置进行重新读取。这意味着，已经开启的曾经由于多次失效而被关闭的服务可以通过强制 `HA Monitor` 脚本刷新配置来重新启用。

20.1.3 潜在的陷阱

像这样的系统有很多陷阱。最坏的情况是两个进程会在同一时间关闭。在这种级别的群集中，对于这种情况无能为力。更高级的群集系统可以将进程直接挂到操作系统内核中来应付各种极端情况，但这里介绍的这个简单脚本只是尽力让进程保持在活动状态。

另一个关于系统配置的问题是放弃与重启失败进程之间的平衡。如果进程总是因为配置问题、频繁的内存泄漏或者其他代码问题而导致失败，则起不到偶尔能让外部用户使用服务的功能，只是不断地掉线。这时最好让服务关闭，并使用人工干预来解决背后的问题。

在这种情况下，存储最近服务故障的时间戳的数组能提供更准确的问题诊断方法。如果在过去的 3 分钟内出现了两次故障，但这之前的故障是在 3 个月以前，那么有理由再重启服务。然而，如果服务不断出现故障，可能直接关闭它更好。对保存故障时间戳的数组进行乱序应当是很容易实现的。

20.1.4 脚本结构

该系统的基本结构在 HA Monitor 中，但它的简易版本嵌在 Friar Tuck 进程中。脚本的另一个关键之处在于用到的数据结构。我们将在下面的内容中逐个介绍。

1. 数据结构

主脚本 hamonitor.sh 充分利用数组来跟踪它能监控的无数量限制的进程的不同方面。Friar Tuck 脚本不需要做这些，并且要简单得多，但使用数组使 HA Monitor 脚本更加灵活。因为 shell 不能处理多维数组，所以进程的每个方面都必须用自己的数组来存储。脚本没有使用 stopcmd、min 与 max 数组；将它们列出来是为了表现完整性，并暗示我们可以考虑其他一些通过足够简单的修改就能完成的任务。pid 数组跟踪进程使用的 PID。如果在收集到 PID 之前有延迟，则数组元素被赋值为负数。

两个脚本都使用的另一个变量是 tag。logger 用它识别正在运行的进程。脚本通过 logger -t "Stag" 实现对自身的精确识别。"Stag" 中的两个引号非常重要。冒号在 tag 之后，所以如果不用引号就成了 friartuck: (8357)/friartuck.sh FINISHING。对比起来，它不如 friartuck (8357): friartuck.sh FINISHING 明了。

```
Apr 20 10:00:20 goldie friartuck (8357): ./friartuck.sh FINISHING. Signalling 8323
to do the same.
Apr 20 10:00:21 goldie hamonitor (8323): /etc/ha/hamonitor.sh FINISHING
Apr 20 10:00:25 goldie friartuck (8357): ./friartuck.sh FINISHED.
```

这对于脚本的调试与诊断特别有用。如上所示，我们很容易看到 friartuck.sh 正在向 PID 8323(即 hamonitor.sh 脚本)发送信号。

2. Friar Tuck

Friar Tuck 进程进行全盘控制，包括启动监控系统与向 HA Monitor 发送信号。Friar Tuck 捕获信号 3 与信号 4，并创建/tmp 文件与 HA Monitor 进行通信。这样可以在发送 SIGQUIT(3)时对两个进程的关闭进行协调或在发送 SIGILL(4)时让 HA Monitor 重新读取配

置文件。

如果 `pgrep` 发现 `hamonitor.sh` 没有以 `root` 身份运行, 则运行该脚本。如果已经在运行, 但不是预想中的 `PID`, 则将该信息记录到日志中并关注新的 `PID`。然而, 如果 `hamonitor.sh` 不断出问题, 则不采取特殊的操作。`friartuck.sh` 总是会尝试对其进行重启。这样比 `hamonitor.sh` 更简单, 但这就是 `friartuck.sh` 需要用来保证 `hamonitor.sh` 一直在运行的原始而简单的方法。

3. HA Monitor

`hamonitor.sh` 脚本大约有 200 行。这在本书中是第二长的单个脚本, 而且大概是结构化 shell 脚本应有的合理长度。一系列简单的命令可能要更长而且还能保持可控性, 但对于比较复杂的、比 `hamonitor.sh` 更长的脚本而言, 则值得将其划分为不同的函数与代码库。

脚本中有一个 `while` 循环。它从中间开始, 一直运行到脚本末尾。尽管 `while` 循环的主要部分是一个内部的 `for` 循环, 但相比 `for` 循环的较具有可控性的长度, 外层的 `while` 循环则很难容纳在一个屏幕高度中。

该脚本可以使用 `bash 4` 中的关联数组。如果 `bash` 不提供关联数组, 配置文件必须命名为 `1.conf`、`2.conf` 等。另外, `declare -A` 语句必须修改为 `declare -a`, 因为 `-A` 是用来声明关联数组的, 并且在 `bash 4` 之前不存在。

脚本以一个 `readconfig` 函数开头。该函数将配置文件逐个读取到数组中。这种方法意味着被监控进程的数目实际是无限的。函数最值得注意的地方是应当从配置文件中读取的变量在读取配置文件之前被删除。如果不这样做, 且某个配置文件漏掉了一个变量, 则该变量会保持先前配置文件定义的值。这样做是错误的, 并且跟踪这种错误非常困难。`readconfig` 还会执行一次初始的检测, 看进程是否正在运行。如果是, 则将 `PID` 数组中的这一元素设置为以指定用户身份运行该进程的 `PID` 列表。

函数 `failurecount` 将进程的 `lastfailure` 数组元素与当前的时间戳进行对比。GNU 的日期格式字符串 `%s` 返回从 1970 年 1 月 1 日到现在经过的秒数。用它来计算两个时间的差很容易。如果时间间隔比允许的故障时间窗小, 则将进程标记为禁用。配置文件中可能有 `STOPABLE=0` 这样一行语句; 它用于 Friar Tuck 进程, 表示无论在任何情况下都不允许进程出现失败。

`startproc` 函数的前两个参数是被开启的标志与运行的用户名。余下的参数用来启动进程。方法是先处理前两个参数, 然后使用 `shift 2` 命令将它们移除。这样就只剩下启动命令及其参数在 `$@` 中。

然后执行 `while` 主循环。该循环分为两部分。第一部分(在 `sleep $DELAY` 一行之前)检查 `STOPFILE` 或 `READFILE` 是否存在。如果存在, 且属主为 `root`, 则脚本要么停止运行, 且删除 `STOPFILE` 来向 Friar Tuck 表示已经接收并处理了该消息; 要么重新读取配置文件, 然后移除 `READFILE`, 这样在下一次循环迭代的时候不用重新读取配置。

在 `sleep $DELAY` 命令之后, HA Monitor 循环遍历 `idx` 数组, 它向循环给出数组每个键的名称。在 `bash 4` 中可以不用 `idx` 数组而用 `${!process[@]}` 来创建这样的列表, 但这里使用 `idx` 列出键更加明了。这种方法也能在没有关联数组的时候使用。

如果服务没有启用(无论是在配置文件中, 或者因为出了太多次的故障), 则将其忽略。

`continue` 回到 `for` 循环来处理下一个进程。

接下来检查 `pid` 数组是否为负数。如果是，则进程最近才启动，而且可能还没有完全完成启动。脚本增加数组元素中的值(越来越接近 0)，并还是使用 `continue` 来处理 `for` 循环中的下一个进程。如果数组元素达到 -1，则使用 `pgrep` 扫描进程树中的运行进程，并将 PID 赋值给 `pid` 数组中对应的进程元素。如果没有找到，则调用 `failurecount` 函数将进程标记为已失败。否则，脚本将新的 PID 记录下来，并在循环接下来的迭代中用于查找。

最后大约 50 行代码是脚本的主要部分。这部分代码对 3 种不同的场景进行了测试。

首先，如果在查询进程时没有找到预想中的 PID，则进程已经失败，必须被重启或标记为已失败。暂时不考虑第二种场景。脚本处理的第三种可能性是同一个进程用之前的 PID 继续运行，并且一切正常。

脚本考虑的第二种可能性是 3 种情况下较为复杂的。如果发现进程正在运行，但 PID 不同，则可能是两种原因：进程已经终止，并且有新进程取代了它；或者除了被监控的进程，还有进程的一些其他实例在运行。

脚本对 PID 列表进行了比较。如果在当前运行的 PID 列表中发现了任何之前监控过的 PID，则增加 `failed` 变量的值。在循环的末尾，`failed` 变量决定了脚本执行的路线。如果没有发现失败的进程，则多余的 PID 也没有害处，且与被监控的进程无关。

如果一个或多个先前监控过的 PID 丢失了，则调用 `failurecount` 函数。这样可以使进程在不久前已经失败过的情况下免于进一步的监控。`pid` 元素被设置为进程的 PID 或者在 `startdelay` 为非零时设置为 $(0 - \text{startdelay})$ 。这可以使进程在需要的情况下稳定下来。

还有一个 `stopha.sh` 脚本。它只是向 `friartuck.sh` 进程(或者是通过命令行传递的其他 PID)发送 3 号信号(SIGQUIT)。Friar Tuck 会捕获该信号，关闭 HA Monitor，然后自行终止。

20.1.5 脚本代码



可从
wrox.com
下载源代码

```
#!/bin/bash

function bailout
{
    logger -t $tag "$0 FINISHING. Signalling $pid to do the same."
    touch /tmp/hastop.$pid
    while [ -f /tmp/hastop.$pid ]
    do
        sleep 5
    done
    logger -t $tag "$0 FINISHED."
    exit 0
}

function reread
{
    logger -t $tag "$0 signalling $pid to reread config."
    touch /tmp/haread.$pid
}
```

```
trap bailout 3
trap reread 4

tag="friartuck ($$)"
debug=9
DELAY=10
pid=0
cd `dirname $0`
logger -t $tag "Starting HA Monitor Monitoring"

while :
do
    sleep $DELAY
    [ "$debug" -gt "2" ] && logger -t $tag "Checking hamonitor.sh"
    NewPID=`pgrep -u root hamonitor.sh`
    if [ -z "$NewPID" ]; then
        # No process found; child is dead.
        logger -t $tag "No HA process found!"
        logger -t $tag "Starting \"`pwd`/hamonitor.sh\""
        nohup `pwd`/hamonitor.sh >/dev/null 2>&1 &
        pid=0
    elif [ "$NewPID" != "$pid" ]; then
        logger -t $tag "HA Process rediscovered as $NewPID (was $pid)"
        pid=$NewPID
    else
        # All is well.
        [ "$debug" -gt "3" ] && logger -t $tag "hamonitor.sh is running"
    fi
done

done



friartuck.sh



#!/bin/bash

function readconfig
{
    # Read Configuration
    logger -t $tag Reading Configuration
    for proc in ${CONFDIR}/*.conf
    do
        # This filename can be web.conf if Bash4, otherwise 1.conf, 2.conf etc
        unset ENABLED START STOP PROCESS MIN MAX STARTDELAY USER STOPPABLE
        index=`basename $proc .conf`
        echo "Reading $index configuration"
        . $proc
        startcmd[$index]=$START
        stopcmd[$index]=$STOP
        process[$index]=$PROCESS
        min[$index]=$MIN
        max[$index]=$MAX
    done
}
```

```

    startdelay[$index]=$STARTDELAY
    user[$index]=$USER
    enabled[$index]=$ENABLED
    idx[$index]=$index
    lastfailure[$index]=0
    stoppable[$index]={STOPPABLE:-1}
    PID=`pgrep -d ' ' -u ${user[$index]} $PROCESS`
    if [ ! -z "$PID" ]; then
        # Already running
        logger -t $tag "${PROCESS}`is already running;`\n
            " will monitor ${USER}'s PID(s) $PID"
        pid[$index]=$PID
    else
        pid[$index]=-1
        if [ "$ENABLED" ]; then
            startproc $ENABLED $USER $START
        fi
    fi
done
logger -t $tag "Monitoring ${idx[@]}"

# Set defaults
DELAY=10
FAILWINDOW=180
debug=9
. ${CONFDIR}/ha.cfg
}

# If Bash prior to version 4, use declare -a to declare an array
declare -A process
declare -A startcmd
declare -A stopcmd
declare -A min
declare -A max
declare -A pid
declare -A user
declare -A startdelay
declare -A enabled
declare -A lastfailure
declare -A stoppable
# Need to keep an array of indices for Bash prior to v4 (no associative arrays)
declare -A idx

function failurecount
{
    index=$1
    interval=`expr $(date +%s) - ${lastfailure[$index]}`
    lastfailure[$index]=`date +%s`
    if [ "$interval" -lt "$FAILWINDOW" ]; then
        if [ ${stoppable[$index]} -eq 1 ]; then

```

```

        logger -t $tag "${process[$index]} has failed twice within $interval"\
            " seconds. Disabling."
        enabled[$index]=0
    else
        logger -t $tag "${process[$index]} has failed twice within $interval"\
            " seconds but can not be disabled."
    fi
fi
}

function startproc
{
    if [ "$1" -ne "1" ]; then
        shift 2
        logger -t "Not starting \"$@\" as it is disabled."
        return
    fi
    user=$2
    shift 2
    logger -t $tag "Starting \"$@\" as \"$user\""
    nohup sudo -u $user $@ >/dev/null 2>&1 &
}

CONFDIR=/etc/ha
tag="hamonitor ($$)"
STOPFILE=/tmp/hastop.$$
READFILE=/tmp/haread.$$
cd `dirname $0`
logger -t $tag "Starting HA Monitoring"
readconfig

while :
do
    if [ -f $STOPFILE ]; then
        case `stat -c %u $STOPFILE` in
            0)
                logger -t $tag "$0 FINISHING"
                rm -f $STOPFILE
                exit 0
                ;;
            *)
                logger -t $tag "$0 ignoring non-root $STOPFILE"
                ;;
        esac
    fi
    if [ -f $READFILE ]; then
        case `stat -c %u $READFILE` in
            0) readconfig
                rm -f $READFILE
                ;;

```

```

*)
    logger -t $tag "$0 ignoring non-root $READFILE"
    ;;
esac
fi
sleep $DELAY
for index in ${idx[@]}
do
    if [ ${enabled[$index]} -eq 0 ]; then
        [ "$debug" -gt "3" ] && logger -t $tag "Skipping ${process[$index]}"\
            " as it is disabled."
        continue
    fi

    # Check daemon running; start it if not.
    if [ ${pid[$index]} -lt -1 ]; then
        # still waiting for it to start up; skip.
        logger -t $tag "Not checking ${process[$index]} yet."
        pid[$index]=`expr ${pid[$index]} + 1`
        continue
    elif [ ${pid[$index]} == -1 ]; then
        pid[$index]=`pgrep -d ' ' -u ${user[$index]} ${process[$index]}`
        if [ -z "${pid[$index]}" ]; then
            logger -t $tag "${process[$index]} didn't start in the allowed timespan."
            failurecount $index
        fi
        logger -t $tag "PID of ${process[$index]} is ${pid[$index]}."
        continue
    fi

    [ "$debug" -gt "2" ] && logger -t $tag "Checking ${process[$index]}"
    NewPID=`pgrep -d ' ' -u ${user[$index]} ${process[$index]}`
    if [ -z "$NewPID" ]; then
        # No process found; child is dead.
        logger -t $tag "No process for ${process[$index]} found!"
        failurecount $index
        startproc ${enabled[$index]} ${user[$index]} ${startcmd[$index]}
        if [ ${startdelay[$index]} -eq 0 ]; then
            pid[$index]=`pgrep -d ' ' -u ${user[$index]} ${process[$index]}`
        else
            pid[$index]=`expr 0 - ${startdelay[$index]}`
        fi
        [ "$debug" -gt "4" ] && logger -t $tag "Start Delay for "\
            "${process[$index]} is ${startdelay[$index]}."
    elif [ "$NewPID" != "${pid[$index]}" ]; then
        # The PID has changed. Is it just new processes?
        failed=0
        for thispid in ${pid[$index]}
        do
            echo $NewPID | grep -w $thispid > /dev/null
            if [ "$?" -ne "0" ]; then

```



```
# one of our PIDs is missing
((failed++))
fi
done
if [ "$failed" -gt "0" ]; then
    failurecount $index
    logger -t $tag "PID changed for ${process[$index]}; was \"\"\\
        \"${pid[$index]}\" now \"${NewPID}\""
    # pid[$index]=-2 #SGP $NewPID
    if [ ${startdelay[$index]} -eq 0 ]; then
        pid[$index]=$NewPID
    else
        pid[$index]=`expr 0 - ${startdelay[$index]}`
    fi
fi
else
    # All is well.
    [ "debug" -gt "3" ] && logger -t $tag "${process[$index]} is running"
fi
done
done
```

hamonitor.sh

```
#!/bin/bash
pid=${1:-`pgrep -u root friartuck.sh`}
kill -3 $pid
```

stopha.sh

```
# Apache is started with apachectl
# but the process is called apache2
START="/usr/sbin/apachectl start"
STOP="/usr/sbin/apachectl stop"
PROCESS=apache2
MIN=1
MAX=10
STARTDELAY=2
ENABLED=1
USER=root
```

apache.conf

```
START="nohup ./friartuck.sh >/dev/null 2>&1"
STOP=/bin/false
PROCESS="friartuck.sh"
MIN=1
MAX=1
STARTDELAY=0
ENABLED=1
```

```
USER=root
STOPPABLE=0
```

friartuck.conf

```
START="sleep 600"
STOP=
PROCESS=sleep
MIN=1
MAX=10
STARTDELAY=0
ENABLED=1
USER=steve
```

sleep.conf

20.1.6 调用结果

只要运行 `friartuck.sh` 脚本就能启动该框架。此处，`/var/log/messages` 文件将发生的事件都记录下来。`sleep` 进程被启动，但发现另外两个要被监控的进程已经在运行。`friartuck.sh` 脚本已经在运行，因为刚刚从命令行启动它。



根据 syslog 的配置方法，本例中的消息可能会记录到不同的文件。

```
# /etc/ha/friartuck.sh
Apr 20 11:03:36 goldie friartuck (10521): Starting HA Monitor Monitoring
Apr 20 11:03:46 goldie friartuck (10521): Checking hamonitor.sh
Apr 20 11:03:46 goldie friartuck (10521): No HA process found!
Apr 20 11:03:46 goldie friartuck (10521): Starting "/etc/ha/hamonitor.sh"
Apr 20 11:03:46 goldie hamonitor (10531): Starting HA Monitoring
Apr 20 11:03:46 goldie hamonitor (10531): Reading Configuration
Apr 20 11:03:46 goldie hamonitor (10531): apache2 is already running;
will monitor root's PID(s) 7663
Apr 20 11:03:46 goldie hamonitor (10531): friartuck.sh is already running;
will monitor root's PID(s) 10521
Apr 20 11:03:46 goldie hamonitor (10531): sleep is already running;
will monitor steve's PID(s) 10273
Apr 20 11:03:46 goldie hamonitor (10531): Monitoring friartuck sleep apache
Apr 20 11:03:56 goldie friartuck (10521): Checking hamonitor.sh
Apr 20 11:03:56 goldie friartuck (10521): HA Process rediscovered as
10531 (was 0)
Apr 20 11:03:56 goldie hamonitor (10531): Checking friartuck.sh
Apr 20 11:03:56 goldie hamonitor (10531): friartuck.sh is running
Apr 20 11:03:56 goldie hamonitor (10531): Checking sleep
Apr 20 11:03:56 goldie hamonitor (10531): sleep is running
Apr 20 11:03:56 goldie hamonitor (10531): Checking apache2
Apr 20 11:03:56 goldie hamonitor (10531): apache2 is running
```

现在可以安全地关闭 `friartuck.sh` 的命令行调用。它会被 `hamonitor.sh` 重启。到这里，整个监控过程在没有控制终端的情况下进行，且每个脚本都可以独立重启另一个脚本。

```
Apr 20 11:04:36 goldie hamonitor (10531): Checking friartuck.sh
Apr 20 11:04:37 goldie hamonitor (10531): No process for friartuck.sh found!
Apr 20 11:04:37 goldie hamonitor (10531): Starting "nohup ./friartuck.sh
>/dev/null2>&1" as "root"
Apr 20 11:04:37 goldie hamonitor (10531): Start Delay for friartuck.sh is 0.
Apr 20 11:04:37 goldie hamonitor (10531): Checking sleep
Apr 20 11:04:37 goldie friartuck (10680): Starting HA Monitor Monitoring
Apr 20 11:04:37 goldie hamonitor (10531): sleep is running
Apr 20 11:04:37 goldie hamonitor (10531): Checking apache2
Apr 20 11:04:37 goldie hamonitor (10531): apache2 is running
Apr 20 11:04:47 goldie friartuck (10680): Checking hamonitor.sh
Apr 20 11:04:47 goldie friartuck (10680): HA Process rediscovered as 10531
(was 0)
```

关闭 `sleep` 进程，然后强迫它按照与 `friartuck.sh` 同样的方式重启。缘于配置文件中定义的配置选项，`sleep` 在 3 分钟之内只能重启一次，否则将其标记为失败。

```
Apr 20 11:05:47 goldie hamonitor (10531): Checking sleep
Apr 20 11:05:47 goldie hamonitor (10531): No process for sleep found!
Apr 20 11:05:47 goldie hamonitor (10531): Starting "sleep 600" as "steve"
Apr 20 11:05:47 goldie hamonitor (10531): Start Delay for sleep is 0.
Apr 20 11:05:47 goldie hamonitor (10531): Checking apache2
Apr 20 11:05:47 goldie hamonitor (10531): apache2 is running
Apr 20 11:05:57 goldie friartuck (10680): Checking hamonitor.sh
Apr 20 11:05:57 goldie friartuck (10680): hamonitor.sh is running
Apr 20 11:05:57 goldie hamonitor (10531): Checking friartuck.sh
Apr 20 11:05:57 goldie hamonitor (10531): Checking sleep
Apr 20 11:05:57 goldie hamonitor (10531): sleep is running
```

在 3 分钟的时间窗内再次将 `sleep` 关闭会导致 `sleep` 被禁用。HA Monitor 脚本直到重新读取配置文件才会再次尝试重启 `sleep`。

```
Apr 20 11:07:08 goldie hamonitor (10531): Checking sleep
Apr 20 11:07:08 goldie hamonitor (10531): No process for sleep found!
Apr 20 11:07:08 goldie hamonitor (10531): sleep has failed twice within 81
seconds. Disabling.
Apr 20 11:07:08 goldie hamonitor (10531): Not starting "sleep 600" as it
is disabled.
Apr 20 11:07:08 goldie hamonitor (10531): Start Delay for sleep is 0.
```

因为 `friartuck.sh` 的配置文件中 `STOPPABLE=0` 这个属性，所以数组值 `${stoppable[friartuck]}` 同样具有 0 值。与 `sleep` 进程不同，Friar Tuck 无论被关闭多少次都会被重启。

```
Apr 20 11:08:38 goldie hamonitor (10531): Checking friartuck.sh
```

```

Apr 20 11:08:38 goldie hamonitor (10531): No process for friartuck.sh found!
Apr 20 11:08:38 goldie hamonitor (10531): Starting "nohup ./friartuck.sh
>/dev/null 2>&1" as "root"
Apr 20 11:08:38 goldie hamonitor (10531): Start Delay for friartuck.sh is 0.
. . . . .
Apr 20 11:09:18 goldie hamonitor (10531): Checking friartuck.sh
Apr 20 11:09:19 goldie hamonitor (10531): No process for friartuck.sh found!
Apr 20 11:09:19 goldie hamonitor (10531): friartuck.sh has failed twice
within 41 seconds but can not be disabled.
Apr 20 11:09:19 goldie hamonitor (10531): Starting "nohup ./friartuck.sh
>/dev/null 2>&1" as "root"
Apr 20 11:09:19 goldie hamonitor (10531): Start Delay for friartuck.sh is 0.

```

初看起来, Apache 似乎是按照同样的方式处理的。通过 `apachectl stop` 命令停止 Apache 会导致其所有进程的终止。HA Monitor 脚本重启 Apache, 正如对 `sleep` 与 `friartuck.sh` 那样。

```

Apr 20 11:12:10 goldie hamonitor (10531): Checking apache2
Apr 20 11:12:10 goldie hamonitor (10531): No process for apache2 found!
Apr 20 11:12:10 goldie hamonitor (10531): Starting "/usr/sbin/apachectl
start" as " root"
Apr 20 11:12:10 goldie hamonitor (10531): Start Delay for apache2 is 2.
. . . . .
Apr 20 11:12:20 goldie hamonitor (10531): Not checking apache2 yet.
. . . . .
Apr 20 11:12:30 goldie hamonitor (10531): PID of apache2 is 11935.
. . . . .
Apr 20 11:12:40 goldie hamonitor (10531): Checking apache2
Apr 20 11:12:40 goldie hamonitor (10531): apache2 is running

```

然而, 如果独立于这些脚本调用 `apachectl restart`, 则 Apache 会以不同的 PID 重新出现, 而这个 PID 不处于 HA Monitor 框架的监视之下。只要上一次故障超出 3 分钟, 就会将这些信息记录下来, 但会继续执行并监控新的 PID。

```

Apr 20 11:17:32 goldie hamonitor (10531): PID changed for apache2; was
"11935" now "12868"
. . . . .
Apr 20 11:17:42 goldie hamonitor (10531): Not checking apache2 yet.
. . . . .
Apr 20 11:17:52 goldie hamonitor (10531): PID of apache2 is 12868.
. . . . .
Apr 20 11:18:03 goldie hamonitor (10531): Checking apache2
Apr 20 11:18:03 goldie hamonitor (10531): apache2 is running

```

在这种状态下, `friartuck.sh` 与 `hamonitor.sh` 可以被安全地关闭, 因为任何一个脚本都会重启另一个。如 20.1.3 节提到的, 如果它们被同时关闭, 可能的情况是无论哪个脚本都不负责另一个脚本的重启, 但可能发生的是在两个进程都被关闭之前一个重启了另一个。

```
Apr 20 11:21:14 goldie hamonitor (10531): Checking friartuck.sh
Apr 20 11:21:14 goldie hamonitor (10531): No process for friartuck.sh
found!
Apr 20 11:21:14 goldie hamonitor (10531): Starting "nohup ./friartuck.sh
>/dev/null 2>&1" as "root"
Apr 20 11:21:14 goldie hamonitor (10531): Start Delay for friartuck.sh
is 0.
Apr 20 11:21:14 goldie hamonitor (10531): Checking sleep
Apr 20 11:21:14 goldie friartuck (13564): Starting HA Monitor Monitoring
. . . . .
Apr 20 11:21:24 goldie friartuck (13564): Checking hamonitor.sh
Apr 20 11:21:24 goldie friartuck (13564): HA Process rediscovered as
10531 (was 0)
. . . . .
Apr 20 11:21:34 goldie friartuck (13564): Checking hamonitor.sh
Apr 20 11:21:34 goldie friartuck (13564): No HA process found!
Apr 20 11:21:34 goldie friartuck (13564): Starting "/etc/ha/hamonitor
.sh"
Apr 20 11:21:34 goldie hamonitor (13620): Starting HA Monitoring
Apr 20 11:21:34 goldie hamonitor (13620): Reading Configuration
Apr 20 11:21:34 goldie hamonitor (13620): apache2 is already running;
will monitor root's PID(s) 12868
Apr 20 11:21:34 goldie hamonitor (13620): friartuck.sh is already
running; will monitor root's PID(s) 13564
Apr 20 11:21:34 goldie hamonitor (13620): Starting "sleep 600" as
"steve"
Apr 20 11:21:34 goldie hamonitor (13620): Monitoring friartuck sleep
apache
Apr 20 11:21:44 goldie friartuck (13564): Checking hamonitor.sh
Apr 20 11:21:44 goldie friartuck (13564): HA Process rediscovered as
13620 (was 0)
Apr 20 11:21:44 goldie hamonitor (13620): Checking friartuck.sh
Apr 20 11:21:44 goldie hamonitor (13620): friartuck.sh is running
Apr 20 11:21:44 goldie hamonitor (13620): PID of sleep is 13638.
Apr 20 11:21:44 goldie hamonitor (13620): Checking apache2
Apr 20 11:21:44 goldie hamonitor (13620): apache2 is running
```

因此，尽管两个进程都被关闭，操作还是一如既往。值得注意的是，这时的 `sleep` 已经在启动新的 `hamonitor.sh` 时重启了。被禁用的状态只存储在运行中的 `hamonitor.sh` 脚本的数组中。很容易对 `hamonitor.sh` 进行修改来将更新的状态写入 `sleep.conf` 文件中，或其他一些包含已有状态的跟踪文件。然而，接下来的测试是禁用一条命令，然后强制 `hamonitor.sh` 重新读取配置文件。这会以相同的方式重新开启这条命令。两次关闭 `Apache` 可以实现这样的测试。

```
Apr 20 11:24:36 goldie hamonitor (13620): Checking apache2
Apr 20 11:24:36 goldie hamonitor (13620): No process for apache2 found!
Apr 20 11:24:36 goldie hamonitor (13620): Starting "/usr/sbin/apachectl
start" as " root"
Apr 20 11:24:36 goldie hamonitor (13620): Start Delay for apache2 is 2.
```

```

. . . . .
Apr 20 11:24:46 goldie hamonitor (13620): Not checking apache2 yet.
. . . . .
Apr 20 11:24:56 goldie hamonitor (13620): PID of apache2 is 14192.
. . . . .
Apr 20 11:25:06 goldie hamonitor (13620): Checking apache2
Apr 20 11:25:06 goldie hamonitor (13620): apache2 is running
. . . . .
Apr 20 11:25:26 goldie hamonitor (13620): No process for apache2 found!
Apr 20 11:25:26 goldie hamonitor (13620): apache2 has failed twice
within 50 seconds. Disabling.
Apr 20 11:25:26 goldie hamonitor (13620): Not starting "/usr/sbin/
apachectl start" as it is disabled.
Apr 20 11:25:26 goldie hamonitor (13620): Start Delay for apache2 is 2.
. . . . .

```

向 `friartuck.sh` 发送 `kill -4` 信号导致 Friar Tuck 创建文件 `/tmp/haread.$pid`，其中 `$pid` 就是 `hamonitor.sh` 脚本的 PID。`hamonitor.sh` 脚本下次循环时会检测到该文件，并重新读取配置文件。

```

Apr 20 11:27:27 goldie hamonitor (13620): Skipping apache2 as it is
disabled.
Apr 20 11:27:35 goldie friartuck (13564): ./friartuck.sh signalling 13620
to reread config.
Apr 20 11:27:35 goldie friartuck (13564): Checking hamonitor.sh
Apr 20 11:27:35 goldie friartuck (13564): hamonitor.sh is running
Apr 20 11:27:37 goldie hamonitor (13620): Checking friartuck.sh
Apr 20 11:27:37 goldie hamonitor (13620): friartuck.sh is running
Apr 20 11:27:37 goldie hamonitor (13620): Checking sleep
Apr 20 11:27:37 goldie hamonitor (13620): sleep is running
Apr 20 11:27:37 goldie hamonitor (13620): Skipping apache2 as it is
disabled.
Apr 20 11:27:37 goldie hamonitor (13620): Reading Configuration
Apr 20 11:27:37 goldie hamonitor (13620): Starting "/usr/sbin/apachectl
start" as " root"
Apr 20 11:27:37 goldie hamonitor (13620): friartuck.sh is already
running; will monitor root's PID(s) 13564
Apr 20 11:27:37 goldie hamonitor (13620): sleep is already running;
will monitor steve's PID(s) 13638
Apr 20 11:27:37 goldie hamonitor (13620): Monitoring friartuck sleep
apache
. . . . .
Apr 20 11:27:47 goldie hamonitor (13620): PID of apache2 is 14711.
. . . . .
Apr 20 11:28:07 goldie hamonitor (13620): Checking apache2
Apr 20 11:28:07 goldie hamonitor (13620): apache2 is running

```

最后，为了关闭整个框架，向 `friartuck.sh` 发送 `kill -3` 信号使 Friar Tuck 创建 `/tmp/hastop.$pid`，然后等待 `hamonitor.sh` 将其删除。随后两个脚本会干净地退出，且不会被重启。`hastop.sh`

脚本也可以实现关闭功能。

```
Apr 20 11:29:35 goldie friartuck (13564): ./friartuck.sh FINISHING.  
Signalling 13620 to do the same.  
Apr 20 11:29:38 goldie hamonitor (13620): Checking friartuck.sh  
Apr 20 11:29:38 goldie hamonitor (13620): friartuck.sh is running  
Apr 20 11:29:38 goldie hamonitor (13620): Checking sleep  
Apr 20 11:29:38 goldie hamonitor (13620): sleep is running  
Apr 20 11:29:38 goldie hamonitor (13620): Checking apache2  
Apr 20 11:29:38 goldie hamonitor (13620): apache2 is running  
Apr 20 11:29:38 goldie hamonitor (13620): /etc/ha/hamonitor.sh FINISHING  
Apr 20 11:29:40 goldie friartuck (13564): ./friartuck.sh FINISHED.
```

20.1.7 小结

高可用性是一个复杂的领域，而复杂度通常会耗费财力。如果某个应用程序的不可用性导致企业的损失，则应当投入资金来减轻这样的故障。我们将获得更完善的系统。它可以监控存储与网络资源，重启应用程序，以及在节点之间切换以防止完全的硬件故障。

然而，当我们面对一个不可靠的应用程序(它在任何时候都不可能手动重启)时，本章介绍的脚本将会让应用程序尽量为我们运行并处理一些会阻碍运行的基本问题，如高可用性脚本自身的失效。另外，该脚本理解与使用起来也足够简单。不用一周的训练时间，我们就能够使用其核心功能。

第 21 章

国 际 化

国际化经常被认为是只有过度设计编程环境中的复杂功能才能完成的任务。其实并非如此，而且写出国际化的 shell 脚本也很容易，如本章的脚本所示。对人类语言的程序化处理总是很有技巧性，并且一次处理多种不同语言更加复杂。关键的两点是让 shell 语法与消息字符串保持分离，并注意每个复数情况。在英文脚本中，`I can see 1 aircraft` 与 `I can see 2 aircraft` 都可以，但在其他语言中，单词 `aircraft` 可能有不同的复数形式。

21.1 实用脚本 21-1：国际化

大多数 shell 脚本是用美式英语编写的，而且其中绝大部分都没有被翻译成其他语言。类似地，一些用母语编写的脚本从没有翻译成其他任何语言。有时候翻译是绝对有必要的——对于只使用一种语言的工作团队的内部脚本而言，其实没有使用其他语言的必要。然而在其他时候，我们可能会遇到很多不同的问题，除非脚本被翻译过了。实际上，如果不理解脚本的内容，则无法使用脚本。如果脚本可以用我们的母语进行通信，则能更清楚地理解其中的信息。另外，对于某些语言的支持可能是法律或合同上的要求。

本章的讨论假设原始脚本用英文编写，并将其翻译成其他语言，但这不是强制性的。原始脚本可以编写成现实中的任何语言；无论脚本输出什么都会变成 `msgid`，然后翻译成 `msgstr`。

我编写的第一个脚本使用了国际化。该脚本对 GNU/Linux 下一个特殊的连接了 USB 的 ADSL 调制解调器进行配置(<http://speedtouchconf.source-forge.net/>)。首先，该脚本只显示一些调制解调器所需的本地设置，用于用户的 ISP——每个 ISP 都有自己的用于 ADSL 的 VPI/VCI 对。早期 ADSL 用户应当知道这些(或运行 ISP 的只能在 Windows 下运行的软件。这样的软件对配置值进行了硬编码)。所以法国的 Wanadoo 使用 8/35，其他法国 ISP 使用 8/67，所有的英国 ISP 都使用 0/38 等。很快这样的调制解调器被全世界广泛技术领域的人们所使用，而且这些人对于英文的了解程度并不相同。除了收集所有这些 VPI/VCI 对，我开始接到对脚本本身进行翻译的请求。最终，志愿者将脚本翻译成丹麦语、西班牙语、法

语、意大利语、挪威语、波兰语、葡萄牙语以及斯洛文尼亚语。这些可以使用 GNU 的 `gettext` 工具来完成——甚至要更简单。这样让不具备技术技能的翻译者进行翻译比较容易，意味着招募志愿翻译者也比较容易。翻译者使用 `gettext` 甚至不需要阅读 `shell` 脚本本身，而只看需要翻译的字符串。这样一来就需要看脚本中哪些地方使用了这样的消息字符串，而 `gettext` 可以提供消息的上下文，并允许程序员用注释将消息标记给翻译者。

21.1.1 用到的技术

- 国际化(i18n)
- 本地化(L10n)
- `gettext`
- `eval_gettext`
- `eval_ngettext`
- `xgettext`
- `msgfmt`

21.1.2 概念

翻译过程分成 4 个步骤。首先对字符串进行国际化，然后翻译(可能译成很多不同的语言)，再进行编译，最后自动本地化为运行时需要的目标语言。`Internationalization` 通常缩写为 `i18n`，`localization` 缩写为 `L10n`。这是因为这两个单词写起来太长；18 表示 i 与 n 之间省略的 18 个字符，10 表示 L 与 n 之间省略的 10 个字符。

国际化是为脚本在世界范围内(国际性的)使用做准备的过程。它涉及对所有需要翻译的字符串进行标记。在 `shell` 脚本使用 `gettext` 的情况下，它确保编程规范让 `gettext` 较为容易地识别出字符串中的变量，并正确地翻译这些变量。

第二步的翻译要获取 `shell` 脚本输出的已有字符串，然后将它们翻译成不同的人类语言。如果开发人员能够流利地将脚本翻译成另一种语言，则在很短的时间内可以很容易地完成所有测试。如果不流利，则使用 `gettext` 重新编写代码，翻译字符串，对翻译进行编译，然后执行测试，这样可能比较麻烦且费时。理想情况就是让翻译者保证最终结果与期望的一致。一个较好的权宜之计是创建一种虚拟的语言。在 20 世纪 90 年代后期，Red Hat Linux 安装程序将 `Redneck` 作为一种语言用在安装过程中。这是一种北美的开发人员可以在测试时使用的语言，而且不需要开发人员熟悉外国语言(政治上有误导性的文本可能会进入这样的翻译中，因为按照 `Redneck` 的定义，它是跨语言/文化的，所以不要冒犯他国的文化)。



Red Hat Linux 5.1(注意它与 RHEL 5.1 不同)安装指南在第 37 页有一处脚注对 `Redneck` 语言进行了如下解释：`Redneck` 语言表示的是 Red Hat Software 的 Donnie Barnes 所说的一种美式方言，并位于为安装程序添加国际化支持的测试用例中。使用 `Redneck` 仅仅是为了体现娱乐价值(并显示要与 Donnie 对话有多么困难)。

然后是第三步。一旦翻译了字符串，则利用与 `gettext` 一起使用的 `msgfmt` 工具将独立的语言文本文件翻译成二进制 `.mo` 文件。这些 `.mo` 文件可以放到 `$TEXTDOMAINDIR` 目录下与区域设置有关的子目录中。根据当前区域设置，我们将使用这些子目录中适当的文件。

文本域告诉 `gettext` 要使用哪个翻译集。如果系统安装了两个不同的应用程序，且它们对于英文短语 `No Change` 都有各自的翻译，则自动售货机程序的翻译(意思是“不找零”)与状态跟踪程序的翻译(意思是“状态与上次检查的相同”)不同。文本域确保使用的是正确的翻译。

第四步也是最后一步为本地化。这一步骤发生在用户的计算机上，条件是脚本被执行，且控制权传递给 `gettext` 用以直接得到显示给用户的相关本地化文本。在 GUI 系统中可能意味着菜单、对话框以及弹出消息是用适当的语言显示的。对于 `shell` 脚本而言，文本会显示为正确的语言。

21.1.3 潜在的陷阱

本脚本有一个不那么明显的陷阱，并且完全不是技术上的。即随着脚本的演化，每次脚本修改如果产生新的或不同的输出，就需要重新翻译。这会产生很大的影响，尤其是对已完成翻译的语言的部分。如果所有的语言已经翻译完成，则对脚本的一处修改可能破坏所有的语言。更糟糕的是那些只有部分支持的语言。

在以商业目的驱动的项目中，如果支持了 5 种语言，5 位译者可能需要对新的文本进行重新翻译。与译者协商的时间与花费对比他们将要翻译的文本可能更多。社区项目需要同样的努力，尽管可能是译者不再对脚本维护感兴趣，这样就必须找到该语言的替代志愿者，否则该语言的翻译就会不完整。如果要支持 50 种语言，则可能需要 50 位译者。在不考虑译者动机的情况下，这样可能出现的问题会是 5 位译者的 10 倍。

从更技术的角度看，人们注意到的第一个问题是 `echo` 用换行符结束输出，而 `gettext` 不是，所以直接将 `echo` 语句替换为 `gettext` 不会有相同的结果。依赖于上下文，通常最简单的解决方案是在 `gettext` 之后加上一个 `echo` 来向输出插入换行。在其他时候，如果使用另外一些像 `printf` 这样更灵活的输出工具，我们可以得到比单独使用 `echo` 时更加简洁、易懂的代码。

更重要的是，`gettext` 并不知道 `shell` 可使用的结构。这还不至于非常糟糕。也就是说译者无须知道关于 `shell` 脚本编程的一切。从这个角度看对于判断什么适于或不适于传递给 `gettext` 是有帮助的。然而，`gettext` 还有一个兄弟程序 `eval_gettext`，它可以对简单的变量语法进行求值，这样变量的值就可以进入可翻译的字符串中。它可以处理简单的变量，如 `$a` 或 `${a}`，但不能像 `${a:-1}` 这样复杂。它不能处理其他的语法结构，如 ``pwd``。所有这样的精简运算都必须在 `eval_gettext` 之外完成。这会给开发人员一些小小的额外负担，但也意味着非技术译者能更容易地进行翻译。

该脚本中的 `for` 循环较短，它使 2、4、6 加倍，用命令 `ans=`expr $i * 2`` 将答案放到 `$ans` 中。`$ans` 随后被传递给 `eval_gettext`，但 `$i` 与 `$ans` 中的美元符号必须进行转义才不会被 `shell` 扩展。传递准确的包括美元符号的文本非常重要，因为 `msgid` 是 `Twice $i is $ans`，而不是 `Twice 2 is 4`。

另外还有单复数的问题。一些语言对于一些特别的单词使用相同的单复数形式。例如

英语中的 `sheep`，可以同时表示单复数。与之相反的是，`child` 的英文复数是 `children`，不是 `childs`。这种结构没有逻辑可循，且不可编程实现。所以翻译时必须看它是否为复数。一些语言对语法的修改完全取决于使用的是单数还是复数。`eval_ngettext` 可以处理这样的差异，如脚本末尾的 `I have n child[ren]` 所示。

21.1.4 脚本结构

该脚本只是一个简单的 33 行 shell 脚本。它创建了一个备份目录 `~/savedfiles`，然后向用户表示欢迎，回显两个随机数，再将 2、4 与 6 乘以 2。最后，脚本宣布有 1 个、2 个或 3 个小孩。没有什么特别复杂的地方，但用于演示国际化已经足够了。而且将脚本本身尽量保持简单有助于将精力集中在翻译上。

首先，`xgettext` 找出脚本中所有可以翻译的字符串，然后创建 `messages.po`。它用适当的格式包含了要翻译的所有字符串。`messages.po` 的副本被用于每种要翻译的语言。看精力的多少，开发人员可能会添加上大多数甚至全部的头信息，然后译者会读取每个 `msgid` (英文文本)，然后将其 `msgstr` 加入目标语言中。这在该脚本中是通过 `vi po/de/script.po` 表示的，其中的 `po` 是用 `$TEXTDOMAINDIR` 变量指定的目录。译者可能将 `msgfmt` 命令当成自己测试的一部分来运行，或者在翻译提交之后由开发人员运行，又或者译者与开发人员都会去运行。该命令创建 `$TEXTDOMAIN.mo` 二进制文件，其中包含了翻译信息。出于演示的目的，这里的文本域是 `mynicescript`，而脚本本身名为 `script.sh`。两者的名称可以相同或者不同，但这里强调它们的不同以示区别。按照惯例，文本域一般都是项目的名称，其中的单个脚本可能只是一小部分。

21.1.5 脚本代码



可从
wrox.com
下载源代码

```
steve@goldie:~/script$ cat script.sh
#!/bin/bash
. gettext.sh
export TEXTDOMAIN=mynicescript
cd `dirname $0`
export TEXTDOMAINDIR=`pwd`/po

savedir=`gettext "savedfiles"`
mkdir ~/.$savedir

gettext "Hello, world!"
echo
gettext "Welcome to the script."
echo

###i18n: Thank you for translating this script!
###i18n: Please leave $RANDOM intact :-)
eval_gettext "Here's a random number: \$RANDOM"
echo
eval_gettext "Here's another: \$RANDOM"
echo
echo
```

```

for i in 2 4 6
do
    ans=`expr $i \* 2`
    eval_gettext "Twice \$i is \$ans"
    echo
done

```

script.sh

首先，`xgettext` 从脚本中提取文本。它创建一个 `messages.po` 文件，其中包含应当由开发人员与译者合作完成的头信息。其中还有一个模板，模板包含了所有在脚本中找到的字符串，由译者进行翻译。



可从
wrox.com
下载源代码

```

steve@goldie:~/script$ xgettext --add-comments='##i18n' script.sh
steve@goldie:~/script$ cat messages.po
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2011-04-06 19:47+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: script.sh:7
msgid "savedfiles"
msgstr ""

#: script.sh:10
msgid "Hello, world!"
msgstr ""

#: script.sh:12
msgid "Welcome to the script."
msgstr ""

#. ##i18n: Thank you for translating this script!
#. ##i18n: Please leave $RANDOM intact :-)
#: script.sh:17
#, sh-format

```

```
msgid "Here's a random number: $RANDOM"
msgstr ""
```

```
#: script.sh:19
#, sh-format
msgid "Here's another: $RANDOM"
msgstr ""
```

```
#: script.sh:25
#, sh-format
msgid "Twice $i is $ans"
msgstr ""
```

messages.po

```
steve@goldie:~/script$ mkdir -p po/de/LC_MESSAGES ← 将区域设置为德语(DE)
steve@goldie:~/script$ cp messages.po po/de/script.po
steve@goldie:~/script$ vi po/de/script.po
steve@goldie:~/script$ cat po/de/script.po
```

```
# My Nifty Script.
# Copyright (C) 2011 Steve Parker
# This file is distributed under the same license as the PACKAGE package.
# Steve Parker <steve@steve-parker.org>, 2011
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: 1.0\n"
"Report-Msgid-Bugs-To: i18n@example.com\n"
"POT-Creation-Date: 2011-04-06 19:47+0100\n"
"PO-Revision-Date: 2011-05-11 12:32+0100\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: German Translator <de@example.org>\n"
"Language: de\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=iso-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"
#: script.sh:7
msgid "savedfiles"
msgstr "gespeichertendateien"

#: script.sh:12
msgid "Hello, world!"
msgstr "Hallo Welt!"

#: script.sh:13
msgid "Welcome to the script."
msgstr "willkommen, um das Skript"
```

```
#. ##i18n: Thank you for translating this script!
```

```

#. ##i18n: Please leave $RANDOM intact :-)
#: script.sh:16
#, sh-format
msgid "Here's a random number: $RANDOM"
msgstr "Hier ist eine Zufallszahl: $RANDOM"

#: script.sh:17
#, sh-format
msgid "Here's another: $RANDOM"
msgstr "Hier ist eine andere: $RANDOM"

#: script.sh:22
#, sh-format
msgid "Twice $i is $ans"
msgstr "zweimal $i ist $ans"

#: script.sh:31
#, sh-format
msgid "I have $i child."
msgid_plural "I have $i children."
msgstr[0] "Ich habe $i Kind."
msgstr[1] "Ich habe $i Kinder."

```

script.po

编译德语字符串

```

➔ steve@goldie:~/script$ msgfmt -o po/de/LC_MESSAGES/mynicescript.mo
  po/de/script.po
steve@goldie:~/script$ mkdir -p po/fr/LC_MESSAGES ← 将区域设置为法语(FR)
steve@goldie:~/script$ cp messages.po po/fr/script.po
steve@goldie:~/script$ vi po/fr/script.po
steve@goldie:~/script$ cat po/fr/script.po
# My Nifty Script.
# Copyright (C) 2011 Steve Parker
# This file is distributed under the same license as the PACKAGE package.
# Steve Parker <steve@steve-parker.org>, 2011
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: 1.0\n"
"Report-Msgid-Bugs-To: i18n@example.com\n"
"POT-Creation-Date: 2011-04-06 19:47+0100\n"
"PO-Revision-Date: 2011-07-01 16:21+0100\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: French Translator <fr@example.org>\n"
"Language: fr\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=iso-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"
#: script.sh:7

```

```

msgid "savedfiles"
msgstr "fichiersenregistrés"

#: script.sh:12
msgid "Hello, world!"
msgstr "Bonjour tout le monde!"

#: script.sh:13
msgid "Welcome to the script."
msgstr "Bienvenue sur le script."

#. ##!18n: Thank you for translating this script!
#. ##!18n: Please leave $RANDOM intact :-)
#: script.sh:16
#, sh-format
msgid "Here's a random number: $RANDOM"
msgstr "voici un nombre aléatoire: $RANDOM"

#: script.sh:17
#, sh-format
msgid "Here's another: $RANDOM"
msgstr "voici un autre: $RANDOM"

#: script.sh:22
#, sh-format
msgid "Twice $i is $ans"
msgstr "deux fois $i est de $ans"

#: script.sh:31
#, sh-format
msgid "I have $i child."
msgid_plural "I have $i children."
msgstr[0] "J'ai $i enfant."
msgstr[1] "J'ai $i enfants."

```

script.po

编译法语字符串

```

→ steve@goldie:~/script$ msgfmt -o po/fr/LC_MESSAGES/mynicescript.mo po
  /fr/script.po

```

21.1.6 调用结果

注意下面的运行示例，File exists 消息是由 `mkdir` 命令而非脚本报告的。这是 `mkdir` 自身的本地化决定的。对于下面这些运行测试，我们保证目录已经存在(用 3 种语言)来演示这样的行为。脚本有目的地将目录名 `savedfiles` 翻译成本地语言。如果脚本决定了目录名都要用英文，则德语消息会是“`mkdir: kann Verzeichnis »/home/steve/.savedfiles« nicht anlegen: Die Datei existiert bereits`”。这样的决定是针对每个文件的。出于明显技术上的原因，不可能仅仅因为区域设置是匈牙利语就将 `/etc/hosts` 重命名为 `/stb/házigazdák`。然而，这

个保存到用户的主目录下的文件目录更适合于翻译，尽管它取决于应用程序，以及对于同一个用户用不同的区域设置重新运行脚本时的处理方法。

```

steve@goldie:~/script$ ./script.sh
mkdir: cannot create directory `/home/steve/.savedfiles': File exists
Hello, world!
Welcome to the script.
Here's a random number: 17365
Here's another: 28848

Twice 2 is 4
Twice 4 is 8
Twice 6 is 12
I have 1 child.
I have 2 children.
I have 3 children.
steve@goldie:~/script$ export LANG=de_DE
steve@goldie:~/script$ ./script.sh
mkdir: kann Verzeichnis »/home/steve/.gespeichertendateien« nicht anlegen: Die Date
i existiert bereits
Hallo Welt!
willkommen, um das Skript
Hier ist eine Zufallszahl: 16618
Hier ist eine andere: 5870

zweimal 2 ist 4
zweimal 4 ist 8
zweimal 6 ist 12
Ich habe 1 Kind.
Ich habe 2 Kinder.
Ich habe 3 Kinder.
steve@goldie:~/script$ LANGUAGE=fr_FR
steve@goldie:~/script$ export LANGUAGE
steve@goldie:~/script$ ./script.sh
mkdir: impossible de créer le répertoire « /home/steve/.fichiersenregistrés »: Le f
ichier existe
Bonjour tout le monde!
Bienvenue sur le script.
voici un nombre aléatoire: 4944
voici un autre: 26037

deux fois 2 est de 4
deux fois 4 est de 8
deux fois 6 est de 12
J'ai 1 enfant.
J'ai 2 enfants.
J'ai 3 enfants.

```


21.1.7 小结

国际化可以是一个很复杂的话题。但实际上，真正的翻译就是一个简单的文本文件中简单的源语言与目标语言对。如果对译者有帮助，开发人员甚至可以很容易地将 `msgid` 与 `msgstr` 字符串删除，然后在翻译结束后将它们放回原处。所需要的只是一个简单的文本文件。

```
msgid "Hello, world!"  
msgstr "Bonjour tout le monde!"
```

复数可能带来额外的复杂度，而且编写的脚本需要让 `msgid` 字符串灵活地结合到 `gettext` 与 `eval_gettext` 要求的简单模式中。所有这些都是为了给译者提供便利，所以这些限制实际上是有好处的。

更大的挑战可能在于招募与激发译者，以及当脚本信息发生变化后让他们对翻译进行更新。在自由软件或开源项目中，保证潜在的译者能很顺手地完成要求他们完成的任务是保留与鼓励志愿者的重要部分。我们可以让译者对项目的翻译工作非常便利，即使是每次翻译一个字符串。甚至可以建立一个网站，让临时的译者选择一种语言，并向他们给出所选语言的一些当前未被翻译的字符串，然后让他们提供翻译。我们甚至可以去掉所有烦人的认证或检查。如果某个字符串接收到了 30 个提交，其中 28 个都是相关的，那么除非是故意为之，否则这些翻译很可能都是正确的，甚至不需要进一步的检查。

部分翻译也可能是一个选择。如果完全翻译是一个达不到的目标，那么脚本可以暂时在部分翻译的情况下进行发布。结果是一些信息会用本地语言显示，而其他的则会用英文显示。这通常要比完全没有语言支持要好。

第Ⅳ部分

参 考 信 息

- 附录 补充材料
- 术语表

附录

补充材料

`man` 与 `info` 页面中有很多关于本书介绍的所有软件的信息，尤其是 `bash` 的 `man` 页面与参考指南，以及 `coreutils` 包的 `info` 页面。然而，这些都是非常密集的文档，内容精确但对于新手来说不是特别有用。

要想更好地了解大多数这些主题，可以使用很多教程以及其他一些更详细的解释性文档。下面列出了我找到的对我自己比较有用的材料，或许可以为读者在某个特定主题上提供更深入的信息。

shell 教程与文档

`bash` 文档可以从两个地方找到：GNU 网站(gnu.org)与 `bash` 当前的维护者 Chet Ramey 的凯斯西储大学的主页。

- <http://www.gnu.org/software/bash/>
- <http://www.gnu.org/software/bash/manual/bashref.html>
- <http://tiswww.case.edu/php/chet/bash/bashtop.html>
- <ftp://ftp.cwru.edu/pub/bash/FAQ>

Mendel Cooper 的 *Advanced Bash-Scripting Guide* 可以在 webofcrafts.net 网站上找到 PDF 文档。也有 HTML 形式的文档；本附录后面还特别强调了一些链接：

<http://bash.webofcrafts.net/abs-guide.pdf>

Andrew Arensberger 的 Ooblick 站点中有很好的关于 `shell` 的页面，包括两个幻灯片：

<http://ooblick.com/text/sh/>

Philip Brown 的 Bolthole 网站有很多信息，包括 `ksh` 教程：

<http://www.bolthole.com/solaris/ksh.html>

Greg Woledge 的 `bash` 指南包含了很多不错的信息：

<http://mywiki.woolledge.org/BashGuide>

ARNnet 保留了对 Steve Bourne 的珍贵采访：

http://www.arnnet.com.au/article/279011/a-z_programming_languages_bourne_shell_sh/

Dotfiles 提供了很多点文件的示例：

<http://dotfiles.org/>

下面是一些关于 *Advanced Bash-Scripting Guide* 的信息的链接。

- <http://www.faqs.org/docs/abs/HTML/assortedtips.html>
- <http://www.faqs.org/docs/abs/HTML/contributed-scripts.html>

Linux 文档项目包含了很多很好的文档，包括 *Bash Beginner's Guide*。虽然文档名中包含了初学者字样，但涵盖了相当多的关于 shell 编程的内容。

<http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html>

最后，我们自己的 shell 编程教程主要针对 Bourne 兼容的 shell，但偶尔会用到一些 bash 才有的功能。我的博客偶尔会登出一些关于 Unix 与 Linux shell 的特定内容。

- <http://steve-parker.org/sh/sh.shtml>
- <http://nixshell.wordpress.com/>

数组

Greg Woolledge 在他的 *Bash Guide* 中包含了关于 bash 数组的非常有用的信息：

<http://mywiki.woolledge.org/BashGuide/Arrays>

工具

find、sed 与 awk 是一些由 shell 脚本调用的更复杂的工具。下面的链接对这些工具进行了更详细的解释。也有整本讨论 awk 与 sed 语言的书。

find

下面的页面包含一些有用的 find 使用示例。

http://www.kingcomputerservices.com/unix_101/using_find_to_locate_files.htm

sed

Sourceforge 的 sed 站点有一些推荐教程的链接。sed1line.txt 文件也是使用 sed 完成常见任务的快速参考。

- <http://sed.sourceforge.net/sed1line.txt>
- <http://sed.sourceforge.net/grabbag/tutorials/>
- <http://www.faqs.org/faqs/editor-faq/sed/>

Lev Selector 维护了一个非常有用的 sed 页面，其中包含了一些其他 sed 信息的链接：

http://www.selectorweb.com/sed_tutorial.html

awk

IBM DeveloperWorks 站点有一些非常有用的 awk 信息:

<http://www.ibm.com/developerworks/linux/library/l-awk1/>

Greg Goebel 有一个非常有用的 awk 入门材料:

<http://www.vectorsite.net/tsawk.html>

Unix 种类

Bruce Hamilton 的 *Rosetta Stone* 是一个非常棒的资源。正如其网站上说的“what do they call that in this world?”，里面包含了大部分主要的类 Unix 操作系统以及如何用它们来执行常见任务。我们可以在上面搜索一些我们知道的信息(如 iptables)，从而找到其他操作系统上的等价表示(ipf、pfctl 等):

<http://www.bhami.com/rosetta.html>

shell 服务

有两种 shell 服务。传统的一种是可以在服务器上建立自己账户的 shell 主机，我们可以用 ssh 登录。最近有了一种较新的形式，如 <http://anyterm.org/>——<http://simpleshell.com/>也是——这种形式使用 AJAX 在浏览器与服务器的 shell 之间传递文本，将 shell 账户转变为 Web 服务。Fabrice Bellard(QEMU 模拟器的作者)甚至写过一个基于 Javascript 的 486 CPU 模拟器，它在 <http://bellard.org/js-linux> 上运行了一个原生的 Linux。

<http://shells.red-pill.eu/>有很多 shell 提供者，同样的还有 <http://www.egghelp.org/shells.htm>。我本人偶尔使用 silenceisdefeat.com(我的用户名为 [steveparker](http://silenceisdefeat.com))。这些站点提供对一个 OpenBSD 服务器的访问权，需要最少 1 美元的一次性捐赠。

被允许使用其他人的系统当然是一种特权，而且使用条款总会反映这一点。在这样的服务中将自己看成是在别人家里做客非常合适。唯一的区别是我们也应当料想到并接受主人对自己的文件和行为进行审查。发送垃圾邮件、拒绝服务攻击、端口映射、入侵、主机破解、匿名尝试以及所有其他明显对主机突发奇想的破坏，如从内部(例如 Bittorrent、DoS)或外部(例如托管)阻塞主机的网络，或者用掉太多 CPU、内存或其他系统资源，这样的行为对于服务来说都是不允许的。这些大部分都是自动完成的，所以这样的尝试也是徒劳的。

术 语 表

\$ 美元符号在 **shell** 中用于引用变量。在正则表达式中，它还表示文本行的结束。

| 管道符号用于在管道程序中连接命令。

**** 反斜线用于表示之后的字符取字面量，且不扩展。有一些例外：****是反斜线的字面量，所以应当使用****来表示字符串**"\"**。另一个主要的例外是在反斜线之后是换行符的情况。这表示续行，即当前行连接到下一行。这样可以使编写的代码更加清晰易懂。

反斜线在单引号中不会对单引号进行转义操作。**echo 'That's all folks'**不会起作用，**echo 'That\'s all folks'**也不行。要进行转义操作，就必须在引号外面使用****：**echo 'That\'\'s all folks'**。

#! **hash-bang**(也称为 **she-bang**)是文件开头的两个特殊字符，表示它后面的可执行文件是脚本执行时使用的解释器(以及可选参数)。

& 该符号告诉 **shell**，它之前的命令要在后台运行。**shell** 保持在前台，且**\$_**变量会赋值为后台进程的 **PID**。

[表示 **test** 程序。

绝对路径 绝对路径以一个斜杠符号开头。**/etc/hosts** 是表示其所指向的系统中某个特定文件的绝对路径。另外可参见**相对路径**。

别名 别名是命令的缩写。它们在交互式 **shell** 会话中才能使用，而不能用于 **shell** 脚本。

数组 数组是具有多个值的单个变量，且通过索引访问每个值。在 **bash 4** 与 **ksh93** 之前，索引必须是整数。从 **bash 4** 与 **ksh93** 开始，关联数组也可以使用字符串作为索引。

bash **Bourne Again Shell** 的缩写，是很多 **Unix** 与 **Linux** 操作系统的默认 **shell**。

内置命令 这是指内嵌到 **shell** 中的命令。**bash** 手册页中“**shell 内置命令**”下列出了 **bash** 的内置命令。它们主要是像 **cd** 这样的命令，不能作为外部进程(修改目录而不影响当前 **shell** 的进程)运行；或者像 **declare** 这样的命令，定义当前 **shell** 对命名变量的处理方式。类似地，**source(.)**命令必须作为当前运行的 **shell** 进程的一部分才能起作用。

一些 **shell** 内置命令会覆盖一些外部的等价命令。**echo** 就是这样一个内置命令，目的是为了效率。

type 命令在搜索 **\$PATH** 环境变量之前会搜索 **shell** 内置命令，而 **which** 只会在 **\$PATH**

中查找。也就是说，`type kill` 输出 `kill is a shell builtin`，而 `which kill` 输出 `/bin/kill`。

命令替换 将一个命令的输出插入到另一个命令中的行为。有两种形式的命令替换。标准形式使用反引号将命令括起来，表明是命令替换；较新的一种形式使用 `$(cmd)`。命令替换可以嵌套；这时反引号必须使用反斜线转义。下面的代码段给出了两种形式；变量 `周` 围需要用引号来保留输出行之间的换行符。最后一个换行符总是会被删除。

```
$ foo=`ls -l \ `which grep\` /usr/bin/test`
$ echo FOO is: "$foo"
FOO is: -rwxr-xr-x 1 root root 119288 Apr 22 2010 /bin/grep
-rwxr-xr-x 1 root root 30136 Apr 28 2010 /usr/bin/test

$ bar=$(ls -l $(which grep) /usr/bin/test)
$ echo BAR is: "$bar"
BAR is: -rwxr-xr-x 1 root root 119288 Apr 22 2010 /bin/grep
-rwxr-xr-x 1 root root 30136 Apr 28 2010 /usr/bin/test
$
```

编译型语言 编译型语言被写成文本文件，然后由编译器分析生成一个二进制文件。二进制文件由操作系统执行。生成的二进制文件跟编译所在的操作系统与架构有关。

dash Debian Almquist SHell 现在是 Debian 与其他 Debian 的衍生发行版中的默认 shell。它比 `bash` 更加小型化、轻量化，但还是保持 POSIX 兼容性。

设备驱动 处理某类特殊设备实现细节的内核代码。设备驱动程序一般在 `/dev` 目录中。根据操作系统的不同，该目录可以是磁盘上的文件系统或者是虚拟文件系统。设备驱动程序通常是块设备或字符设备。块设备是像磁盘驱动器这样只能逐块写入的设备。大多数其他设备都是字符设备，如终端、音频驱动、内存与网络设备等。另外还有特殊的设备驱动程序，如 `/dev/random`、`/dev/zero` 与 `/dev/null`。它们不关联到任何物理硬件——参见 `null`。

环境 进程的环境是指进程的状态。它包括当前工作目录、打开的文件、子进程，以及为进程设置的环境变量。

FIFO 先入先出的管道。可以在第 14 章中找到 FIFO 的例子。这是由文件系统中的一项表示的数据，一般会关联到存储在物理设备中的数据，尽管对于像 `/proc` 与 `/dev` 这样的虚拟文件系统而言情况不一样。文件还包含存储在索引节点中的元数据，并有一个存储在目录项中的名称。

FSF 自由软件基金会，由 Richard M. Stallman 博士建立，是 GNU 项目的主要赞助机构。GNU 项目也由 Stallman 博士建立。

函数 函数就是代码块，但在定义时(尽管 shell 在分析函数时会报告任何语法错误)不执行，而是 shell 的额外可用命令，并且可以调用。第 8 章详细介绍了函数。

GNU GNU's Not Unix 项目将大多数原始 Unix 工具重写并扩展为自由软件。

here 文档 here 文档使用 `<<` 语法向命令提供标准输入。它的主要用途是向命令提供多行输入，而不必先将这些行写入文件，然后从该文件重定向。我们可以在 `<<` 之后定义一个定界符；一行中的定界符表示输入的结束。下面的代码演示了 here 文档的用法。

```
$ cat - > /tmp/output.txt << END_IT_HERE
> hello
> this is a test.
> END_IT_HERE
$ cat /tmp/output.txt
hello
this is a test.
$
```

here 字符串 here 字符串的语法是<<<；它与 here 文档类似，只是<<<之后不是定界符，而是要执行的命令。如下面的代码所示，文本按字面进行接收，但可以使用命令替换提供命令的输出。为了保留命令输出中的换行符，我们必须将整个表达式放到双引号中。脚本只是将输入的两行读取到\$foo 与\$bar 中，然后将它们显示到标准输出。

```
$ cat /tmp/herestring.sh
#!/bin/bash
read foo
echo Foo is $foo
read bar
echo Bar is $bar
$ /tmp/herestring.sh <<< ls
Foo is ls
Bar is
$ /tmp/herestring.sh <<< `ls /tmp`
Foo is chris.txt herestring.sh keyring-vcxP9t MozillaMailnews orbit-steve
sh-thd-1305760934 ssh-pQhaCK2245 virtual-steve.NqnWUy
Bar is
$ /tmp/herestring.sh <<< "`ls /tmp`"
Foo is chris.txt
Bar is herestring.sh
$
```

无限递归 参见*递归*，*无限*。

inode 文件系统中的每个文件都有一个索引节点(inode)。该节点存储与文件本身有关的一些关键元数据，包括指向文件内容所在位置的连接。存储在 inode 中的关键数据如下：

- 属主
- 属组
- 权限
- 文件大小
- 连接数目
- 上一次的改变时间(ctime)、修改时间(mtime)与访问时间(atime)

这样的结构意味着一个文件可以出现在多个目录中和/或具有多个名称；文件的每个“副本”不需要额外的磁盘空间(除了目录项)，而连接数会增加。当所有的副本都被删除时，连接数会被减小到 0，且可以释放空间。然而，如果文件在上一个连接删除的时候处于打开状态，则 inode 会反映出这一情况。文件在被删除时，对于将文件打开的进程保持

在可用状态，直到所有这样的进程都关闭了文件。

解释型语言 解释型(与编译型相对)语言每次分析并执行一行代码。这样导致的一个副作用是，语言通常也可以交互式地使用。用解释型语言编写的程序可以在支持该语言(尽管系统之间微妙的差别会增加一些复杂度)的任何系统上运行。`shell` 就是这样的语言。对比请参见**编译型语言**。

内核 操作系统的核心。内核的启动要早于任何其他程序；它对硬件拥有完全控制权，包括(对于 80386 中的 x86 架构或更新的架构)独特的将 CPU 转换为保护模式的能力。这为抢占式多任务、中断服务与内存管理提供了可能性。

ksh Kornshell 由 David Korn 编写，是 AT&T Unix 的一部分。它现在是一个开源项目，而且是很多 GNU/Linux 发行版与各种 Unix 版本的一部分。

Linux 类 Unix 操作系统内核，最早由 Linux Torvalds 开发，现在还在其管理之下。

null NULL 字节是 ASCII 码 0。`/dev/null` 是“位桶”——它会丢弃任何发送给它的内容。如果从中读取，则不会有任何输出。相反，`/dev/zero` 提供一个恒定的 NULL 字符流。这两个 `/dev` 设备都是设备驱动程序的特例，因为它们并没有提供到某个特定硬件之间的接口。

进程 操作系统的执行项。每个进程都有一个进程 ID(PID)，而且在 `/proc/PID` 中也有一项，其中包含进程的状态，如进程打开的文件。在 `shell` 脚本中，`shell` 是一个进程，它在自身内部执行内置命令。外部命令(如 `grep`)会生成一个新的进程。新进程会运行，并设置返回码。返回码由 `shell` 的 `$?` 变量获取。

递归，无限 参见**无限递归**。

重定向 将一个文件(最常见的是输入或输出流，如 `stdin` 或 `stdout`)的内容发送到另一个文件的行为。第 10 章介绍了重定向。

相对路径 相对路径不以斜线开头。`../etc/hosts` 指向 `etc` 目录中的 `hosts` 文件，而 `etc` 目录处于当前运行进程的父目录中。`etc/hosts` 指向 `etc` 目录中的 `hosts` 文件，而 `etc` 目录就在当前运行进程所在目录之中。

sh 默认的系统 `shell`。通常都为 Bourne `shell`，或者其他 POSIX 兼容的 `shell`。对于 `bin/sh` 具体功能的混淆可能在 `shell` 可移植性方面导致较大问题。

shell `shell` 是 Unix 与 Linux 系统的默认环境、命令解释器与编程语言。它是用户与内核之间的接口。

标准输入(stdin)、标准输出(stdout)与标准错误(stderr) 标准输入、输出与错误是所有进程都会打开的 3 个文件描述符。这 3 个文件描述符分别为 0、1 与 2。`echo hello` 会显示到标准输出；`echo error >&2` 会显示到标准错误。

Unix 一种多用户、多任务的企业级操作系统。最早于 1969 年开发，而且一直都非常常用。Unix 是 The Open Group(<http://opengroup.org/>)的商标。

空白字符 空格、制表符与换行符都被归为空白字符。在默认情况下，内部字段分隔符(SIFS)被赋值为这 3 个字符。